1. Introduction to Quartus and Circuit Diagram Design
2. A Quartus Project from Start to Finish: 2 Bit Mux Tutorial
3. Lab 1-1: 4-Bit Mux and all NAND/NOR Mux
4. Lab 3-1 Basic MSP430 Assembly from Roots in LC-3
5. Lab 3-2 Digital Input and Output with the MSP430
6. Lab 4-1 Interrupt Driven Programming in MSP430 Assembly
7. Lab 4-2 Putting It All Together
8. Lab 5-1 C Language Programming through the ADC and the MSP430
9. Lab 5-2 Using C and the ADC for "Real World" Applications with the MSP430
10. Helpful General Information
    1. MSP430 LaunchPad Test Circuit Breadboarding Instructions
    2. A Student to Student Intro to IDE Programming and CCS4

Introduction to Quartus and Circuit Diagram Design
Introduces FPGAs and the means of programming them.

Circuit Design in ELEC 220

For the first lab and first project of ELEC 220 we will be focusing on the creation of circuit diagrams using Altera's Quartus II Web Edition. In addition to simulating these circuits on a computer, we will also be configuring Field-Programmable Gate Arrays (FPGAs) from these diagrams. Our target platform will be Terasic's DE0 Development and Education Board which uses Altera's Cyclone III FPGA chip.

FPGAs

An FPGA is an integrated circuit, composed of many logic elements, which can be reconfigured by the user to reproduce a variety of circuits. Each logic element contains several different logic gates and memory elements which can be used to recreate a wide variety of circuit components. The DE0 board we will be using has over 15,000 logic elements although we will only be using less than 1% of these, even for the calculator project. Hopefully, after completing this project you will have a better understanding of the power and versatility of FPGAs.

FPGA Configuration

In order for an FPGA to emulate a desired circuit, it must first be set to the proper configuration specified in a data file uploaded to the board. This data file is created, and often uploaded to the board, using FPGA design software such as Altera's Quartus II. By providing Quartus with information regarding the specific FPGA to be configured, a Quartus project can be easily replicated on an FPGA, shortening delays between concept and protype stages in designing circuits.

HDL vs. Schematic Diagrams

There are two ways to specify the intended function of a Quartus project. The more straightforward method is to simply create a schematic diagram of the desired circuit as though you were drawing it out on paper or building it on a breadboard. This has the advantage of being very easy to grasp, however, it requires you to work out the logic for the entire circuit and lay out all the components. The other more abstracted method is to use a Hardware Description Language (HDL) such as Verilog. Writing using this specification language allows you to specifiy the intended function of the circuit from which Quartus creates an optimized circuit layout. Although this method does not give you as much individual control over the design, it allows you to more easily go from concept to end product by tasking your computer with the bulk of the design work. However, for simplicities sake, we will be using schematic diagrams for the majority of this course, with HDL files provided to you for use in later designs.

Course Overview

The first few labs are designed to give you practice in using Quartus to create schematic diagrams by tasking you with creating schematics for circuits you are already familiar with. They also provide a brief review of the inner workings of components which will be used extensively in later designs. We will then move onto more complex circuits when we introduce the idea of a clock signal as an input to a circuit and design finite state machines. Finally, this section of the lab will culminate in the design of a simple calculator, though it’s not quite like most simple calculators you may be used to. This will draw on previous lessons on circuit design, finite state machines, and clock signals while also providing an introduction to simple computers that will carry on into the next protion of the lab section.

A Quartus Project from Start to Finish: 2 Bit Mux Tutorial
Describes the creation of a project using Altera's Quartus II 11.0, simulating with the Altera University Program Simulator, and programming the DE0 board from Terasic.

Building Projects in Quartus

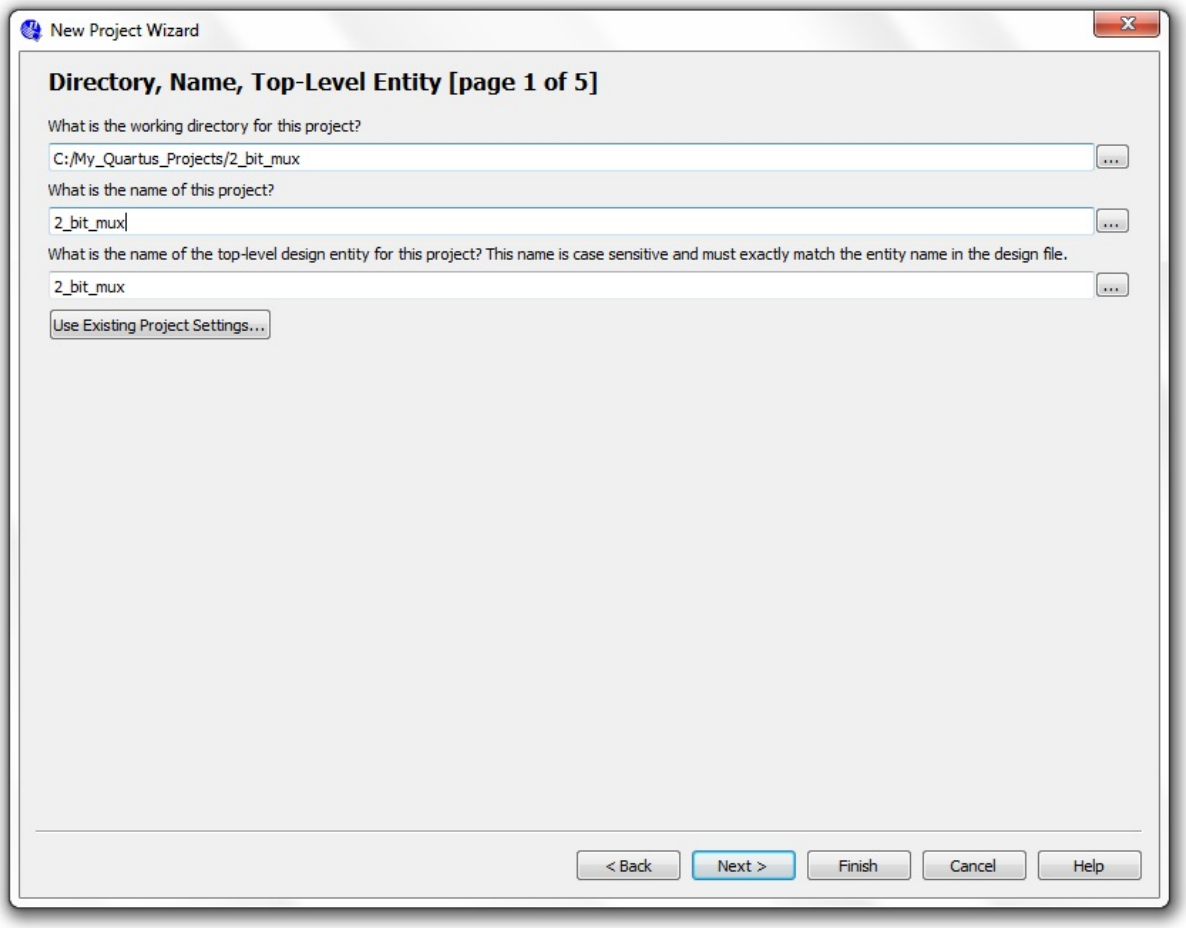
This section is intended to provide an in-depth introduction to creating projects in Quartus, laying out a circuit diagram, simulating the circuit, and finally using the project to configure an FPGA through an example project showcasing a 2-bit MUX. Altera has made a very nice tutorial for Quartus as well which you can find here. Altera's tutorial is meant for a different board than the DE0 we will be using so make sure to account for that. Also, they have a slightly different method for connecting inputs and outputs to the FPGA. Either method works and you can use whichever one you prefer, however, the method set forth in this section will likely be more straightforward and user-friendly. Additionally, you can access another tutorial from within Quartus at any time by clicking on Tutorial in the Help menu.

Starting a Quartus Project

A Quartus project acts as a support structure for a collection of design files. It serves to bring them together in a common working environment, define their relationships both within the project to each other and to the FPGA, and define common characteristics. All work in Quartus starts with a project.

- Begin by opening Quartus II Web Edition. A screen titled "Getting Started with Quartus II Software" should open from which you can select Create a New Project. Otherwise select File->New Project Wizard. Make sure you select this and not simply New, which would instead create a new file.
- In the working directory field specify the folder,"My_Quartus_Projects" for the purpose of this example, to save

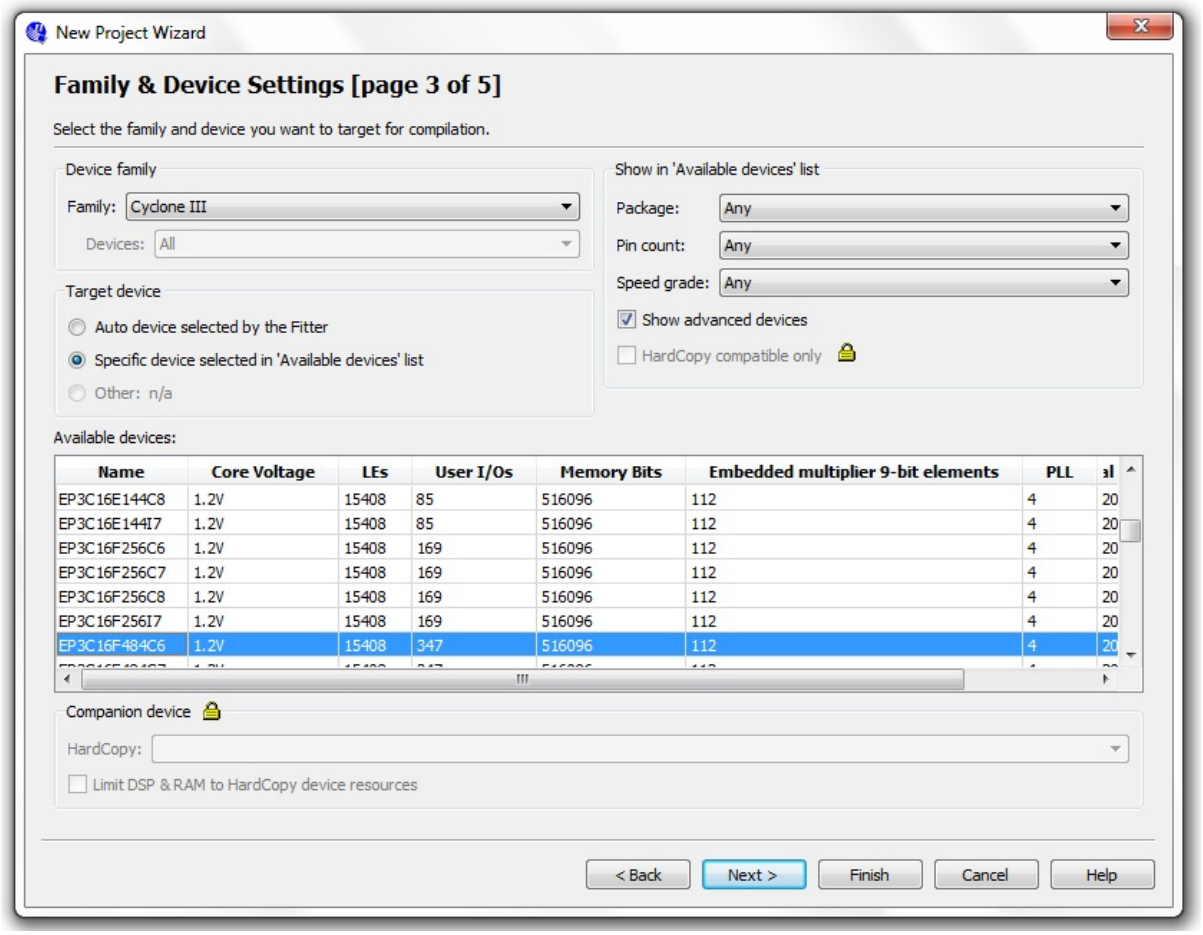
your project in. While you can make this folder on your U: drive, Quartus will generally run faster if working on projects in the C: drive. It is recommended to make temporary folder on the C: drive to put your projects in and transfer them to your U: drive for safe keeping. Note that Quartus will not create a folder for the project files in this location, it will merely save the files here so make sure the lowest level folder is somewhere set aside for this particular project. This will make it easier to locate files in the project and to transfer the project between different computers. Finally, enter the desired name for your project, the final field for the top level design file name will fill itself in as you name the project. It is recommended for simplicities sake that the project and the folder it's in have the same name. Also note that Quartus will not let you use spaces in your naming, underscores or dashes are recommended instead. The name "2_bit_mux" will be used for the purposes of this example.



Specifiying a Project Location and Name

- Next you will see the Add Files screen. All of the labs and projects you will be working on will either have all necessary files included or be started from scratch so we won't be using this feature for now. It is also possible to add files whenever you open a file or save as and we will want to do this during this tutorial in order to ensure our project works as expected.

- After this you will have to specify your target FPGA. The FPGA in the DE0 board we will be using is a **Cyclone III EP3C16F484C6**. You can also find this information by looking at the specification printed on the chip itself.



Selecting the correct FPGA

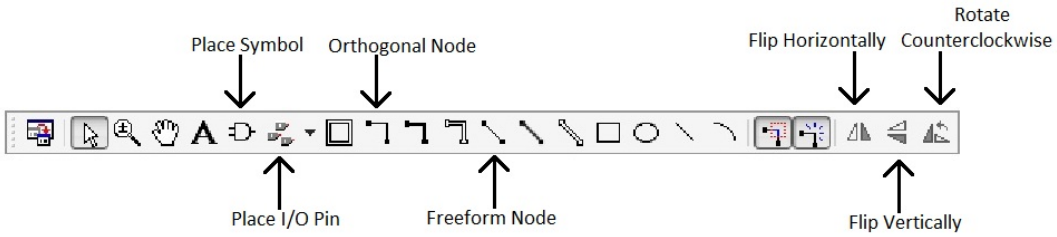
- The next screen allows you to specify other programs to use with this project in addition to Quartus. We won't be using any of these so just click next. After reviewing everything on the final screen to make sure it's set up as you want it and you're ready to begin laying out your circuit.

Building a Circuit in Quartus

- Although we specified a name for our top level design file, we still need to create it. Go to File->New or hit Ctrl+N and select Block

Diagram/Schematic File under Design Files. Once it's open go ahead and Save As, Quartus should automatically give it the same title as the project. Make sure that the box titled "Add file to current project" is checked before saving and that the file is being saved into the project folder.

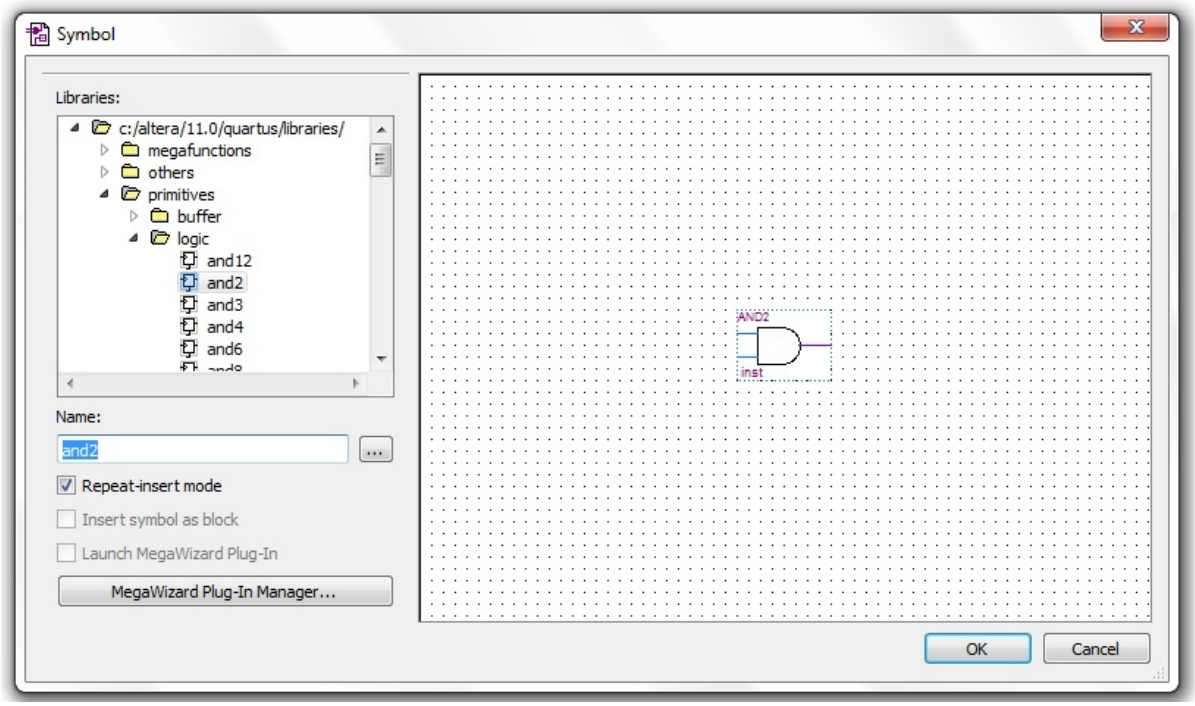
- You should now see a grid of dots and just at the top of it a toolbar. This is where most of our work in Quartus will take place.



The toolbar and some of the tools which will be frequently used.

- In the upper left corner of the window is the project navigator. Since we only have one file in our project, there's not much to see here, but if we had more we would be able to easily keep track of the hierarchy of all the files within the project. Additionally, we can easily open up files associated with this project by double clicking them within this box.
- We'll start by adding symbols to our schematic. Normally you would want to first plan out your circuit design by using Karnaugh maps to write logical functions for the operation of your circuit, however, we'll proceed as though this step has already been completed.
- Click on the place symbol tool to open up the library of available symbols. This can include the default symbols included with Quartus as well as any user created symbols. Within the Quartus library, the majority of the symbols we'll be using will come from the "primitives"

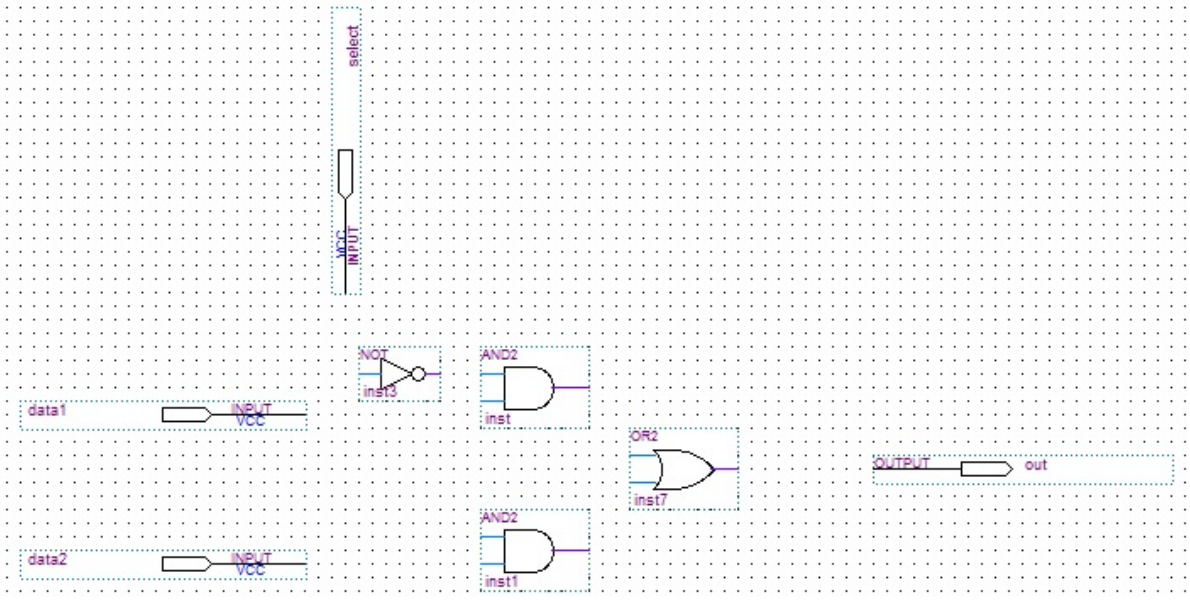
folder. Start by finding a two input AND gate. You can either navigate to the “logic” folder under primitives and find the gate labeled “and2” or simply search for this symbol using the name box below the browser. Note that the name typed here has to exactly match the symbol name for Quartus to find it. Before you click okay, make sure that the box labeled repeat-insert mode is checked as shown below.



The Quartus symbol browser

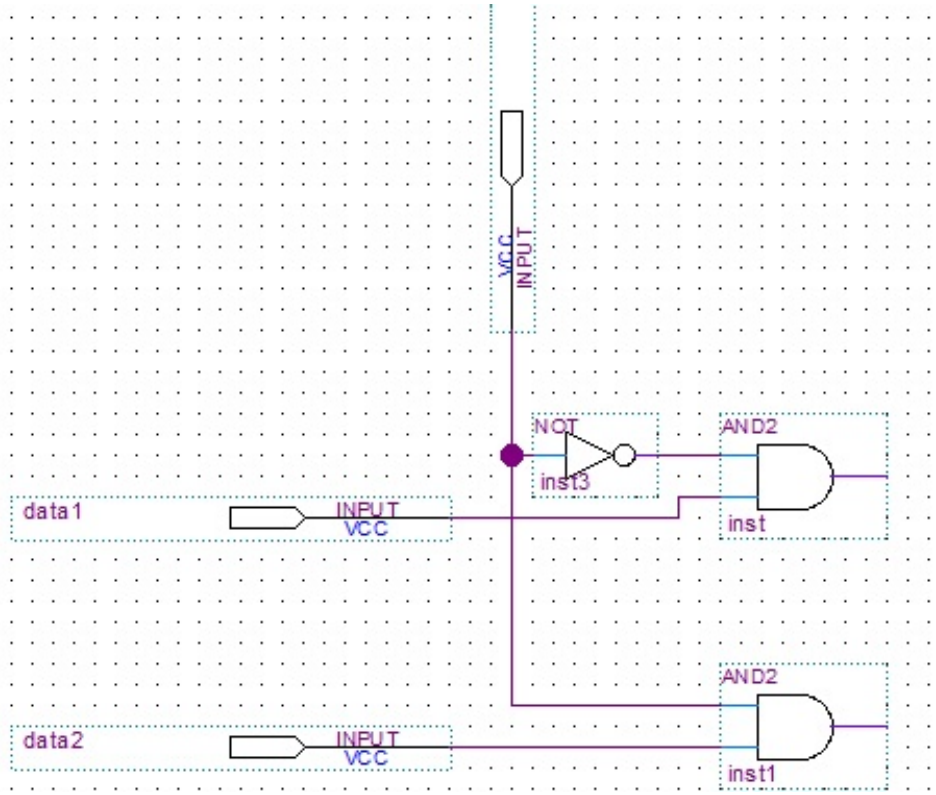
- Place two AND gates onto the grid. Although their relative position isn’t that important since we can remotely connect symbols, it always helps to have a neat circuit layout so for now place them relatively close together. Once you’re done, hit escape to exit from placement mode.
- Continuing on, go back to the symbol browser and select an “or2” gate, also located in primitives->logic. Place one of these gates to the right of your two AND gates.

- Next we'll add an inverter to implement the select logic for the MUX. In the symbol browser find the "not" gate. Place this close to the input of one of the AND gates. Note that it shouldn't be a problem at this point, but if you ever find yourself running out of room on the grid, drag a component to the edge of the screen to expand the available area.
- Now we'll add in I/O pins. This is where signals will enter and leave the schematic. They can be connected to other schematics in the project or connected to inputs and outputs on the board, though we'll define these connections later. For now, go to the drop-down menu on the Pin Tool and choose input. Again, you can place your pins anywhere due to remote wiring, but for now, place two pins to the left of your logic gates for the inputs to the MUX and one above them for the select signal. Go back to the symbol tool, select output, and place one output pin to the right of your circuit for the output of the MUX. Right click on your I/O pins and select properties. From here give each pin a representative name, which will help out in the later I/O assignment phase.



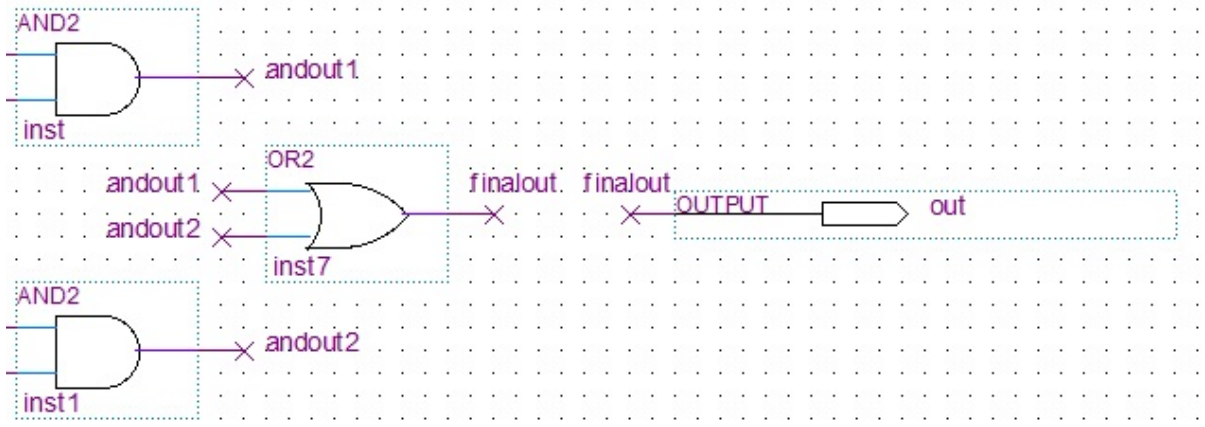
One possible way to layout your gates and pins on the grid

- The final step will be to connect all our components together. As previously mentioned, we can run wires directly between components or make wireless connections.
- To make a wired connection, either select the orthogonal node tool or move your mouse over one of the ports on a symbol, the pointer should change to the look like the node tool. Then click and drag from the origin port to the port you wish to reach and release. Be careful not to intersect any other ports as this will cause them to be joined to the wire, although crossing over other wires will not create a connection. You can tell if a wire is connected to something by the large dot, the typical indicator of connections in circuit diagrams. Go ahead and connect up the inputs to the two AND gates so that they will function as the beginning of a MUX.



Possible circuit wiring for the first stage of the MUX

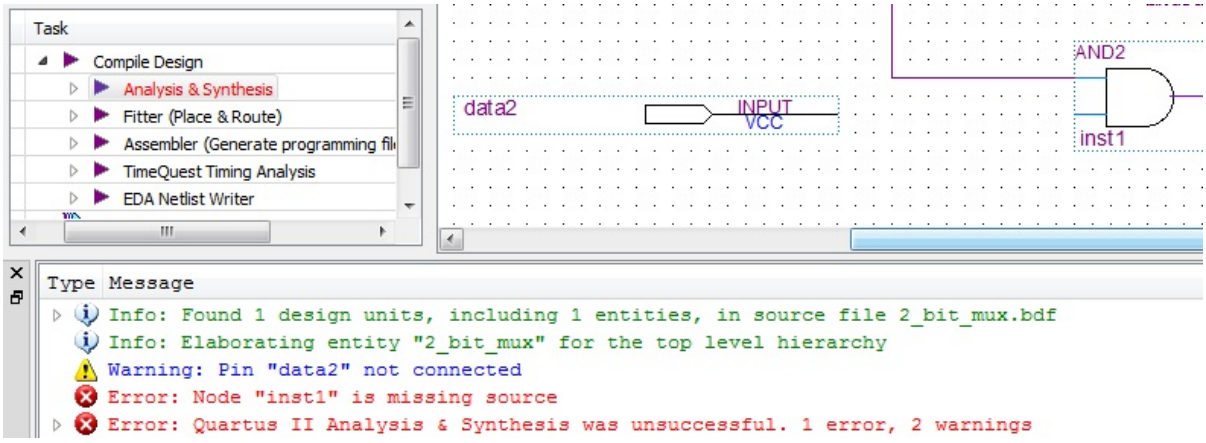
- While this method is fairly straightforward, it has its disadvantages such as possible unintended connections and vast webs of wiring on more complicated circuits. We can simplify the process with wireless connections.
- Unfortunately, you cannot directly name ports. Instead we will connect a small piece of wire to the ports and name this wire. Any wires on the grid which share the same name are connected together.
- Begin by placing short bits of wire at all the remaining ports in the circuit. A length of one on the grid is sufficient though a length of two may be easier to work with. Once placed, right click on the wire and select properties.
- Under the General tab you can enter a name for the selected wire. As with project names, Quartus won't allow for spaces so either remove them or use underscores. To connect any other wire with this named one, simply repeat the procedure. Using this wireless method, connect the remainder of the MUX together.



Example wireless connections

- The final step before we get to pin connections is to make sure our circuit is functional. On the left side of the screen is the Tasks menu

where we can find a variety of commands to create a finished design file. Eventually we will want to compile our whole design, though for now we can simply go through the Anaylsis & Synthesis step. Double click on Analysis & Synthesis for Quartus to check over the circuit for any potential issues such as unconnected ports. If Quartus finds something wrong it will halt the process and display the error in the message box at the bottom of the screen.



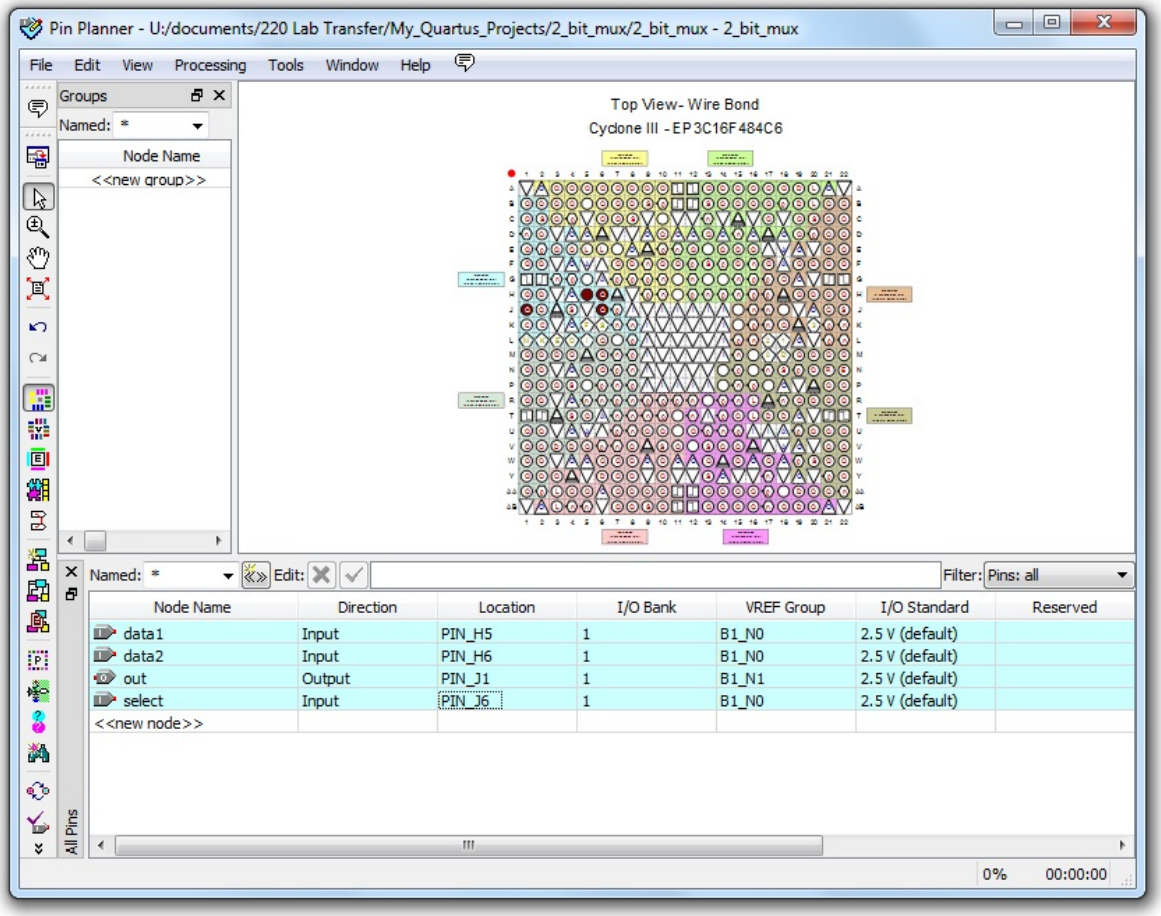
An unsuccessful analysis & synthesis process due to an unconnected port

Defining I/O Connections

- Once we've successfully performed Analysis & Synthesis we are ready to move on to defining pin connections. In order to do this we first need to know the pin addresses of the input and output devices on our DE0 board. These can be found in the DE0 User Manual on pages 24-29, pages 27-32 of the PDF. For now we will only be looking at the switch and LED pin assignments.
- For each entry in the assignment table, the signal name corresponds to the identifier printed on the board next to the relavent device and the

pin name tells us where we should connect to in order to access that device.

- To specify these connections we will use the Pin Planner located under Assignments->Pin Planner. By performing Analysis and Synthesis earlier, we gave Quartus information on how many I/O pins we had on our circuit diagram and what their names were. Now we just need to connect these with pins on the board. By putting in the name of a physical pin under the Location column in the Pin Planner, we tie that point on our board to the specified point on our circuit.
- Although we'll be using a particular pin layout here, you can setup your pin assignments in whatever way you feel works best for you. On future labs/projects pin layouts will already be setup so the labbies and in particular the project graders will be expecting a particular board setup and you should leave assignments as they are.
- For this example we'll use the rightmost slider switch, SW[0], for our select signal. Since we can see in the user manual that SW[0] is tied to PIN_J6, we simply type this, or even just J6 and it will fill in the name, into the Location column next to the "select" listing under the Node Name column. We'll continue in this fashion, assigning "data1" to SW[1] at PIN_H5, "data2" to SW[2] at PN_H6, and assigning "out" to the rightmost LED, LEDG[0] at PIN_J1.

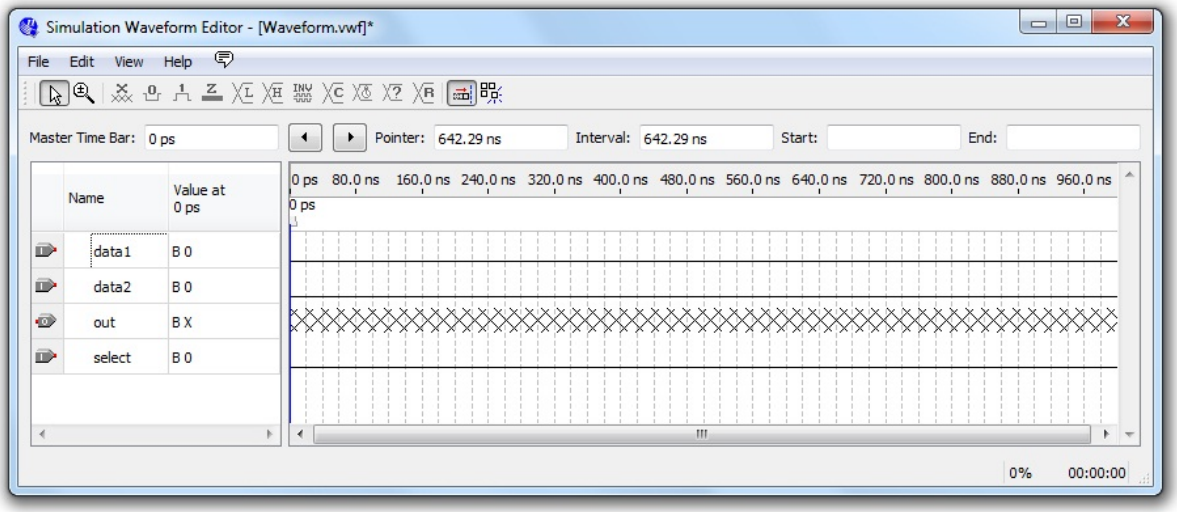


Example pin assignments in the Pin Planner

- Now that we have finished assigning pins, we can go back and run the complete compilation process. Double click on Compile Design in the Tasks menu of the Project Navigator to run all of the sub tasks. Inevitably, you will get warnings about some features not being available without a subscription and there not being a clock, since we didn't need one. Also, you will get critical warnings telling you that a specific design file is needed for the Timing Analyzer. These extra features are not required and these warnings can be ignored.

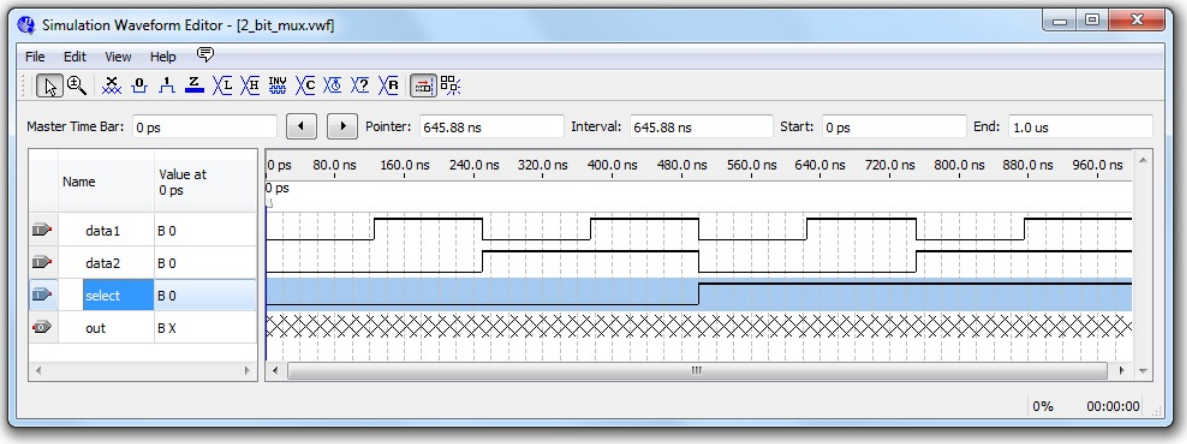
Waveform Simulation

- Before actually programming the FPGA on the board, it is a good idea to simulate a variety of inputs to our circuit and check the responses. Although the ability to simulate inputs to circuits was removed from Quartus II beginning with version 10.0 , these features can still be used with the Altera University Program Simulator.
- Opening the Altera U.P. Simulator should open two windows, the U.P simulator and Qsim. Go to Qsim, select File->Open Project, and select your .qpf project file for the 2 bit mux. Next go to File->New Simulation Input File to open up the Simulation Waveform Editor.
- Right click in the white space under the Name heading and select Insert Node or Bus. From this window click the Node Finder button. Finally, click the List button to have the Waveform Editor import the I/O ports from the project file. Move all of these nodes over to the Selected Nodes box and return to the Waveform Editor Window which should now list these I/O ports along the left side. By clicking and dragging the name of a signal you can rearrange the order they are displayed in, useful for separating the input and output signals. Go ahead and save the waveform file in the project folder for the 2 bit mux.



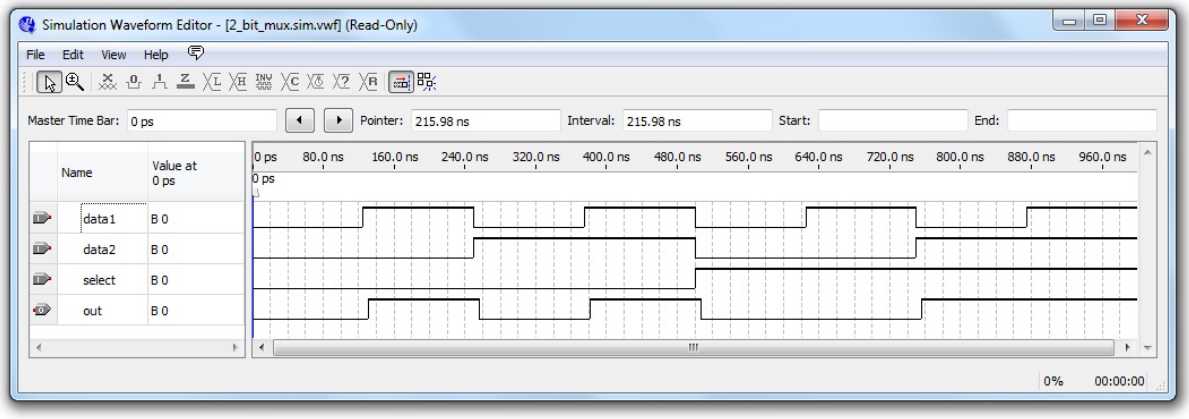
Simulation Waveform Editor Window

- To begin with, all inputs are set to a constant value of 0 and the output is undefined since we have not yet run the simulation. Note the timing intervals displayed along the top. These are not as important now, but will be very useful once we start building project with clocks.
- To change the value of an input, click and drag along a waveform to select one or more intervals. Once selected, you can change the highlighted interval with buttons in the toolbar to set intervals low, high, undefined, opposite of their current value and several other options. For the purposes of testing all possible input combinations, we can either manually set the intervals or use the “Overwrite Clock” button to set up several alternating signals of differing periods.
- For starters select the entire data1 signal by clicking the name and then click the Overwrite Clock button. The Waveform Editor should have defaulted to a total time of 1000ns so set the period of this signal to be 250ns. Select the data2 signal and give it a signal with a period of 500ns and then a signal with period of 1000ns for select. Over the 1000ns of the simulation, this will test all the possible input combinations. Once finished save this waveform file and return to Qsim.



Waveforms to test all input combinations

- Go to Assign->Simulation Settings. The Waveform Simulator supports two modes: Functional, where only the logic of the system is tested and timing is not considered, and Timing, where delays and other timing constraints are taken into account. In order to perfom Functional simulation you must first go to Processing->Generate Simulation Netlist, but for now we'll just do a Timing simulation. In the Simulation Settings box make sure Timing is selected and then browse for the waveform file you created.
- Finally go to Processing->Start Simulation or click the blue arrow over the waveform. The simulator will run and once finished it will open up waveform window containing your specified input waveforms and the resulting output. Once you are satisfied with the results or have made the necessary changes, we can move to the final step, programming the board.



Simulation Output

Programming the Board

- Now that we are certain our project will function as intended, we can program our FPGA. Make sure that the DE0 board is plugged into the

computer and powered on. The DE0 offers two modes of programming: one which retains the program in volatile memory only as long as the board is powered on and another which stores the program in non-volatile memory to be retrieved when the board is powered on. For our purposes the volatile memory storage will be sufficient. To set the board for this programming method, make sure the switch next to the 7 segment display is set to RUN.

- In the Tasks menu below the Analysis & Synthesis and Compile Design commands we used earlier, click on Program Device. Next to Hardware Setup should be listed USB-Blaster [USB-0]. If not, click on Hardware Setup and select USB Blaster from the drop down menu. Make sure that Mode is set to JTAG and that the Program/Configure box next to the .sof file is checked.
- Once ready click Start and wait for the board to be programmed. You can see the state of the programming process in the message bar where it will inform you once it's finished. If you followed the same structure as the tutorial, SW0 should serve as the select switch with SW1 and SW2 toggling the two data inputs high or low. With select in a low state, the mux will take the value from SW1 and in a high state the value from SW2, either of which will be output on LEDG0.
- This concludes the tutorial on Quartus projects. It should now be a simple matter to create a 4-bit mux and move on to the rest of the projects.

Lab 1-1: 4-Bit Mux and all NAND/NOR Mux
Briefly describes the tasks for Lab 1.1 of Rice University's ELEC 220 course.

Lab 1-1

4-Bit Mux

For the first part of Lab 1 you will first design a 4-bit mux, similar to the 2-bit mux covered in the tutorial on Quartus projects found here. An already started Quartus project complete with I/O pins and pin assignments for the DE0 board can be found on OWL Space. This will provide a helpful starting point and ensure a common board setup, making it easier for the labbies to check your circuits' functionality. Download the .zip file and extract it to a temporary working directory on the C: drive, making sure to move it over to your U: drive before you leave the lab.

NAND or NOR only Mux

Create a new version of your original 4-bit mux using only NAND or only NOR gates. A useful tutorial for NAND/NOR conversion can be found here. Do not use inverters (NOT gate) or gates with inverted inputs in your design.

Lab 3-1 Basic MSP430 Assembly from Roots in LC-3
In this lab, students apply what they have learned to implement some basic assembly coding principals on real world hardware. They take what they know from the educational LC-3 and apply the basic principals to a new hardware situation.

An Intro to the MSP430 from the LC-3

This week you will go over the basic differences between the MSP430's assembly ISA and the LC-3's, and learn how to write a basic assembly program for the MSP-430 using TI's Code Composer Studio. You have two main tasks ahead of you:

1. Following the ESCAPE platform labs 0 and 1, setup and establish communication with an ESCAPE sensor board. Run the test program to see if you can communicate wirelessly from the computer and if your sensors are working.
2. Coding in MSP430 Assembly, **implement a Fibonacci sequence calculator**. This should be done with a loop and run infinitely. Step through, explain, and demonstrate the code, using the CCS4 Debugger. Be sure to view the registers while stepping through the program. Observe the amount of CPU cycles each of the instructions takes to complete. Detailed Instructions

Some Background Information

Main Differences Between MSP430 and LC-3

- **The MSP430 has a larger assembly instruction set than the LC-3**

  MSP430 assembly includes some task specific instructions (Such as `inc` and `dec`) to simplify reading the language
  Some MSP430 assembly instructions are interpreted instructions (Such as `pop` and `push`)
  Interpreted Instructions
  An instruction that is decomposed by the assembler into several smaller/ more basic fundamental instructions.

Example:
`pop R3` contains two implicit instructions: `mov @SP, R3` and `add #0x02, SP`

- **Math and logical instructions are similar, but do not have a specific destination.**

  MSP430 instructions come in two flavors, dual operand and single operand. Neither type has an explicit destination register, rather, the last operand serves as the destination too.
  For Example: `add R4, R5` in MSP430 assembly corresponds to `add R5, R4, R5` in LC-3

Note: Be careful to not overwrite data you wish to keep! If you need to preserve the values in both operand registers, you will need to save one of them first using a `mov` instruction.

- **MSP430 Supports some byte as well as word instructions**

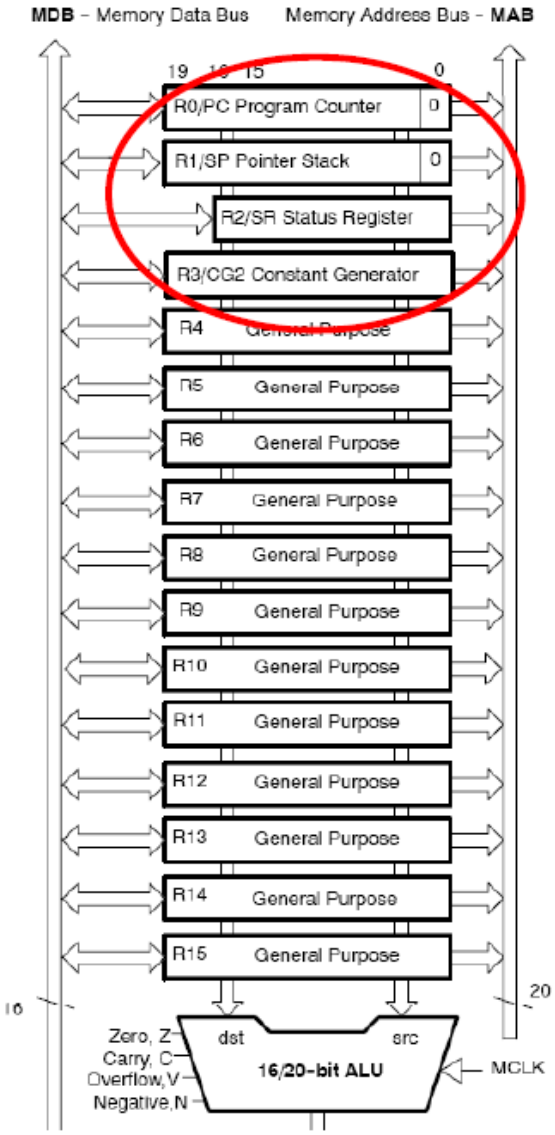
  Some MSP430 instructions allow you to address and write/read from a specific 8 bit byte in memory instead of the entire 16 bit word. The MSP430 memory has byte level addressability, but word instructions only operate on even numbered memory addresses (implicitly modifying the next odd numbered memory byte too). In many cases, especially when working with memory mapped I/O registers, you may need to operate on one specific byte only. To do so, just add a `.b` onto the end of the assembly instruction
  For example: `mov.b #0, &P1DIR` sets 8 bit length P1DIR register to zero without accidentally modifying the registers around it.

Note:MSP430 assembly specifies `.w` for executing word length instructions as well as `.b` for bit length instructions. The assembler

by default assumes word length, so you the programmer don't have to explicitly write `mov.w R5, R14` although you should be conscious that `mov R5, R14` means the same thing.

- **The MSP430 has 16 CPU registers**

  The MSP430 has twice as many CPU registers as the LC-3. Like in the LC-3 though, some of the MSP430's registers are reserved for the MSP430 runtime environment. Registers R0-R3 are reserved (Program Counter, Stack Pointer, Status Register, and a Constant Generation Register respectively), leaving registers R4 through R15 available for general purpose use as defined by the programmer.
  In your assembly programs you have 12 general purpose registers at your disposal, but you also must manage and keep track of the additional options.
  MSP430 Register Usage Diagram



- **Indirect, relative, and absolute addressing occurs differently**

  Instead of different indirect and direct load and store instructions (`LD, LEA, LDI`, etc...), the MSP430 uses one versatile `mov` instruction with different operand addressing modes.
  `mov` can both read and write from memory-- it acts like both a load and store. (`mov R4, &0x0200` corresponds to a `ST` while `mov &0x0200, R4` corresponds to a `LD`) Be careful though, unlike in LC-3, `mov` **does NOT update the condition register.**
  Differentiate between the various direct and indirect modes by using special syntax to specify the type of operand you want. This allows

you to mix addressing types (read indirect and store direct, etc...) even though everything is in one mov instruction.

- Direct register access: Rn (where n is the number of a general purpose register) Example: `R4` refers directly to R4
- Immediate Values: #x (where X is an immediate numerical value or label) Example: `#02h` refers to the literal hex number 2
- Indirect Access From a Register: @Rn (where n is the number of a general purpose register) Example: `@R6` refers indirectly to the data stored in the memory location in R6
- Indirect Offset Access: x(Rn) (where n is the number of a general purpose register and x is either an literal offset or a label) Example: `0(R7)` refers to the data stored in the location in memory pointed to by R7

> **Note:**This has the same end result as `@R7`. By TI code convention though, @Rn cannot be used to specify the destination of an operation, so if you wish to store a result indirectly, you must use the 0(Rn) syntax.

> **Note:**In this example R7 essentially contained the address while the literal offset was a small number. Offset Access can be very powerful when looked at the other way: where the literal contains a starting location in memory (potentially a label) and the register contains a small offset value incremented to access a series of locations in memory.

MSP430 Addressing Modes

Table 3–3. Source/Destination Operand Addressing Modes

As/Ad	Addressing Mode	Syntax	Description
00/0	Register mode	Rn	Register contents are operand
01/1	Indexed mode	X(Rn)	(Rn + X) points to the operand. X is stored in the next word.
01/1	Symbolic mode	ADDR	(PC + X) points to the operand. X is stored in the next word. Indexed mode X(PC) is used.
01/1	Absolute mode	&ADDR	The word following the instruction contains the absolute address. X is stored in the next word. Indexed mode X(SR) is used.
10/–	Indirect register mode	@Rn	Rn is used as a pointer to the operand.
11/–	Indirect autoincrement	@Rn+	Rn is used as a pointer to the operand. Rn is incremented afterwards by 1 for .B instructions and by 2 for .W instructions.
11/–	Immediate mode	#N	The word following the instruction contains the immediate constant N. Indirect autoincrement mode @PC+ is used.

You can also perform indirect or relative operand addressing with operations other than loads and stores

Example:
`add @R4, R5` takes the data stored in the address pointed to by R4 and adds it with R5, storing the result in R5.

For more information, see the summary chart [link] or the comprehensive MSP430 users guide section 3.3.0 through 3.3.7

- **The MSP430 has two types of memory**

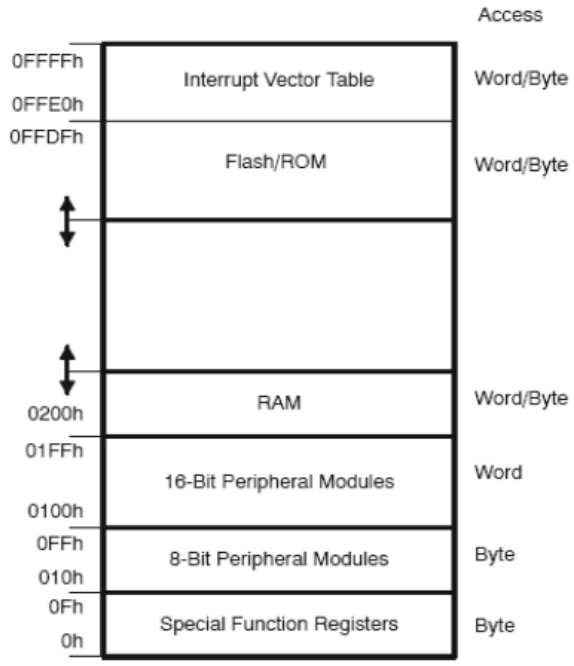
  The MSP430 has both traditional RAM and non-volatile Flash memory. On a power reset, all values in RAM are cleared, so your program will be stored in Flash. The Flash write process is fairly involved, so we won't be writing to it in this class during run time (Code Composer will take care of loading your programs). In a nutshell, your program must store any temporary or changing values

to RAM memory, although it can read your instructions and any preset constants from flash
Important Memory Locations:

0x0200 : The Lowest Address in RAM
0x0280 : The Highest Address in RAM
0xF800 : The Beginning of Flash Memory
0xFFE0 : The Beginning of the Interrupt Vector Table

MSP430 Memory Map

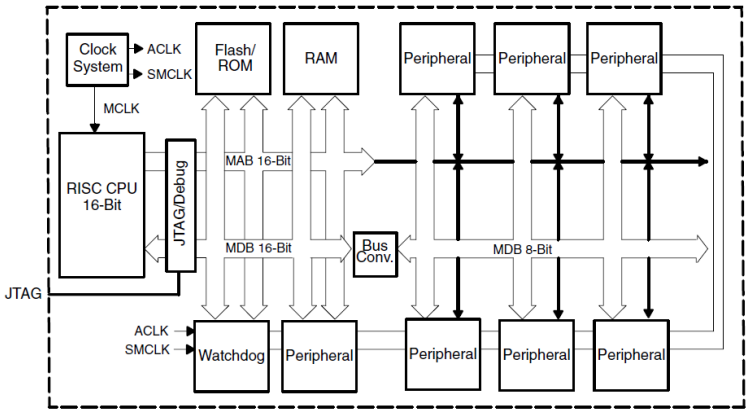
Figure 1–2. Memory Map



The MSP430 Uses Memory Mapped I/O Peripherals

- These devices function independently of the main processor, and use memory mapped registers to communicate with the program executing on the main CPU.
- Peripherals free up CPU resources and also allow more usage of low power CPU suspend modes. You'll learn more about peripherals in Lab 5

MSP430 Architecture Block Diagram



Example Code Translations

LC-3 Assembly	LC-3 Pseudocode	MSP430 Assembly	MSP430 Pseudocode
`AND R4,R5,R6;`	R4 <- R5 & R6	`mov.w R5,R4;` `and.w R6,R4;`	R4 <- R5 R4 <- R4 & R6

LC-3 Assembly	LC-3 Pseudocode	MSP430 Assembly	MSP430 Pseudocode
`BRz R4,Loop;`	if R4 == 0, branch to label "Loop"	`tst R4; jz Loop;`	load the attributes of R4 into the SR jump to label "Loop" if the zero bit is flagged

Other Useful Information

The code composer debugger actually runs on the real MSP430 hardware through a JTAG interface. To debug code, **you have to have the launchpad board plugged into the computer**.

The debugger controls the CPU's clock (and therefore can monitor it). **To see how many clock cycles something takes, go to Target -> Clock -> Enable, and look in the bottom right corner of the screen for a small counter with a clock next to it.**

Part I Assignment Detail

Your task is to create a simple MSP430 assembly program using CCS4 and the MSP430 launchPad to calculate a Fibonacci sequence. You do not need to explicitly display the sequence, but rather use the Code Composer register view tools to watch the sequence progress as you step through your program.

To view the registers in Code Composer Studio v4, first start a debug session. Once you are in the debug perspective, you can go to View--> Registers to open the register dialog. From there, expand the section "Core Registers" to see your CPU registers, or the section "Port_1_2" to see the raw data from the input pins.

Enable the clock cycle monitor (Target-->Clock-->Enable) and you will see a yellow clock icon at the very bottom of your screen. This tells you how many actual CPU clock cycles have passed since you enabled it. Observe the different amounts of time that different instructions take.

The Fibonacci Sequence

The sequence of numbers starting with 0 , 1 in which N= (N-1) + (N-2)

0, 1, 1, 2, 3, 5, 8, 13, 21, 34...

Note: The Fibonacci sequence plays an important role in the natural world. It appears in many biological sequences, and is fundamentally linked to the famed "golden ratio." For more "fun" info about Leonardo Fibonacci, see the ever reliable Wikipedia

Diagrams courtesy of TI document slau144e "MSP430 User's Guide"

The LC-3 was developed by Yale N. Patt (University of Texas at Austin) and Sanjay J. Patel (University of Illinois at Urbana-Champaign) and is used in their book *Introduction to Computing Systems*.

Lab 3-2 Digital Input and Output with the MSP430
This module goes over the basics of digital I/O and the digital interface between a microcontroller and the outside world.

Basic Digital I/O in the Real World

In this lab you'll go over the basics of how to setup and use the GPIO on the MSP430. This will allow you to get data from the outside world, run some processing on it, and then output it again as useful information. You only have one task this week:

1. Coding in MSP430 assembly, **write a simple I/O echo program**. Setup the GPIO pins and poll the input switches for any changes. On a change, take the input and display it to the output. Step through this program to observe how it behaves. Assignment Details

Digital I/O Basics

GPIO

Philosophy

- The MSP430 uses a limited number of GPIO hardware pins that are assignable to several functions depending on your specific model and your program's needs. Our version, the MSP430G2231, can have the Port_1 pins act as digital output, digital input, or ADC input.
- The pins are organized into ports, with each port usually one byte (8 bits/pins) wide. On larger versions of the processor (different format chips with physically many more pins...) you can encounter several ports, but in this lab you will only be using Port_1 and Port_2
- You can set each pin's function independently (input or output) by modifying some memory mapped I/O registers. Since we want to do both, we will divide P1 into half inputs and half outputs as needed.

Usage

- The I/O ports are memory mapped into the top of the MSP430 address space.
- There are several registers associated with each port. For now, you only need to worry about four (P1IN, P1OUT, P1DIR, and P1REN).

P1IN

The P1IN register is located at address `0x0020` in memory, which you can also refer to using the C symbol `&P1IN`
The register holds the values the MSP430 sees at each pin, regardless of the pin direction setting.
To read the register, it is good practice to use a `mov.b` instruction to avoid accidentally reading adjacent registers

> **Note:**If you are looking to test or read just the pins set to input, you will have to mask the P1IN register to zero out the other unwanted/output pins. Reading P1IN reads the entire port, regardless of pin direction.

P1OUT

The P1OUT register is located at address `0x0021` in memory, which you can also refer to using the C symbol `&P1OUT`
If their direction bits in P1DIR are set to output/ "1", the corresponding pins will output the values set in P1OUT.
If a pin's direction bits are set to input in P1DIR and its resistors are enabled in P1REN, P1OUT controls the pin's connection to the pull-up resistor. Setting P1OUT to "1" enables the pull-up, while setting it to "0" leaves the input to float in a high impedance state.
To set P1OUT, use a `mov.b` instruction to set several pins at once. To set individual bits to "1", you can use an `or.b` instruction with a "1" in the positions you want to set. To clear individual bits/ set them to zero, use an `and.b` instruction with mostly "1"s except for a "0" for the bits you want to clear.

P1DIR

The P1DIR register is located at address `0x0022` in memory, which you can also refer to using the C symbol `&P1DIR`
The value of the bits in P1DIR determines whether the MSP430 hardware leaves the pin in a high impedance state where it responds to external voltage changes (which you can read at P1IN), or in a low impedance state where the MSP430 drives the output voltage to a certain value determined by P1OUT.
To set the bit directions all at once, use a `mov.b` instruction, but to change individual bits regardless of the others, use an `and.b` or a `or.b`
Set the corresponding bits to "0" to set pins to input mode, or to "1" to set them to output mode.

P1REN

The P1REN register is located at address `0x0027` in memory, which you can also refer to using the C symbol `&P1REN`
P1REN controls whether the MSP430 Launchpad enables the integrated pull-up resistor for a given pin.
The pull-up resistors allow the use of single pole switches. They prevent the input signals from floating randomly while the switches are open by loosely tying the inputs to Vcc. When the switch is closed though, the much stronger connection to ground wins out, pulling the inputs down to GND.
Set the corresponding bits to "1" to enable a pin's pull-up resistor, or to "0" to disable it (disabled by default).

So What?

In this lab we're going to use the MSP430's GPIO pins, combined with some external switches and an LED display, to build a basic I/O system for our board. Because of how things fit together on the board, it makes sense to use P1.0-P1.3 (the first three Port_1 GPIO Pins) to read the input switches and P1.4-P1.7 for the output signals.

Outputs

Setting up the outputs is easy-- simply set the upper four bits (bits 4-7) of `&P1DIR` to "1", and then write the output to the upper four bits of `&P1OUT`. That means you'll have to shift your data left 4 positions before output, but you should already know a simple technique to do so!

Note: You'll notice that when you change the output, the corresponding input bits also change. This happens because the input hardware always reads the status of the line, regardless if it is set to input our output. Changing the `&P1DIR` values only connects or disconnects the driving circuitry built into the MSP430. In advanced applications this can be used to analyze potential faults in the circuitry outside the chip.

Inputs

Inputs are also "easy," but there are a few hardware concepts you'll need before you understand how they work!

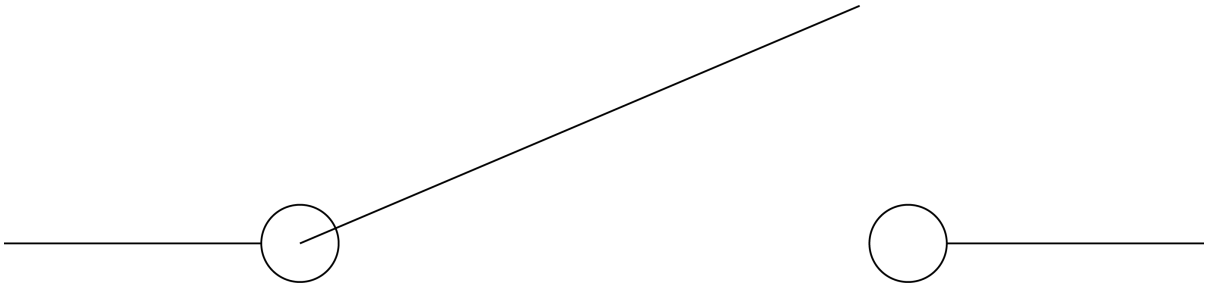
A Little Bit About Wires

As mentioned briefly in class, binary digital logic has two valid states, plus one third mystery state. That third state, "The High Impedance State," (High-Z) just means that the wire isn't connected to anything. You've already talked about using so called tri-state buffers to negotiate who can talk on a shared bus-- the listening components enter the high impedance state, allowing the transmitting component's signal to drive the bus with no conflicts.

Note: Impedance is a generalized form of the classical Resistance concept. Impedances can be real or complex valued, and apply too signals expressed in complex exponential form (**whether constant or variable!**). To learn

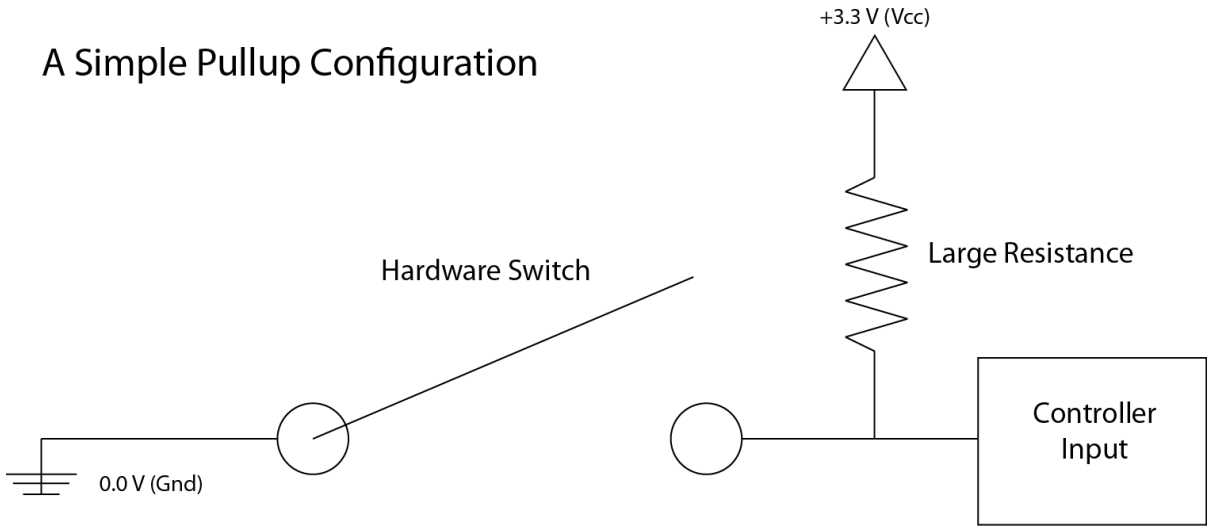
more about impedance, check out Dr. Johnson's sections from the Elec 241 course notes.

A Basic Switch



In order to read useful input from your switches, you need them to be "0" in one state, and "1" in the other. Yet knowing what you know about the third state, the switch shown above will actually give a "0"/"1" (depending on what you connect it to) when closed and "High-Z" when open. Because there's nothing else driving the sensor input besides our switch, **the input value will be random when the switch is open**. In digital logic this is called floating, and it is a very very bad thing.

One simple solution is the **Pull-Up (or Pull-Down) Resistor**. Connecting the floating side of the switch to a logic level through a large resistor will tie down the floating input when the switch is open, but won't effect the read value much when the switch is closed.



As you can see, when the switch is closed, the input is shorted to ground and reads zero. When the switch is open, the pull-up resistor holds the previously floating end at Vcc.

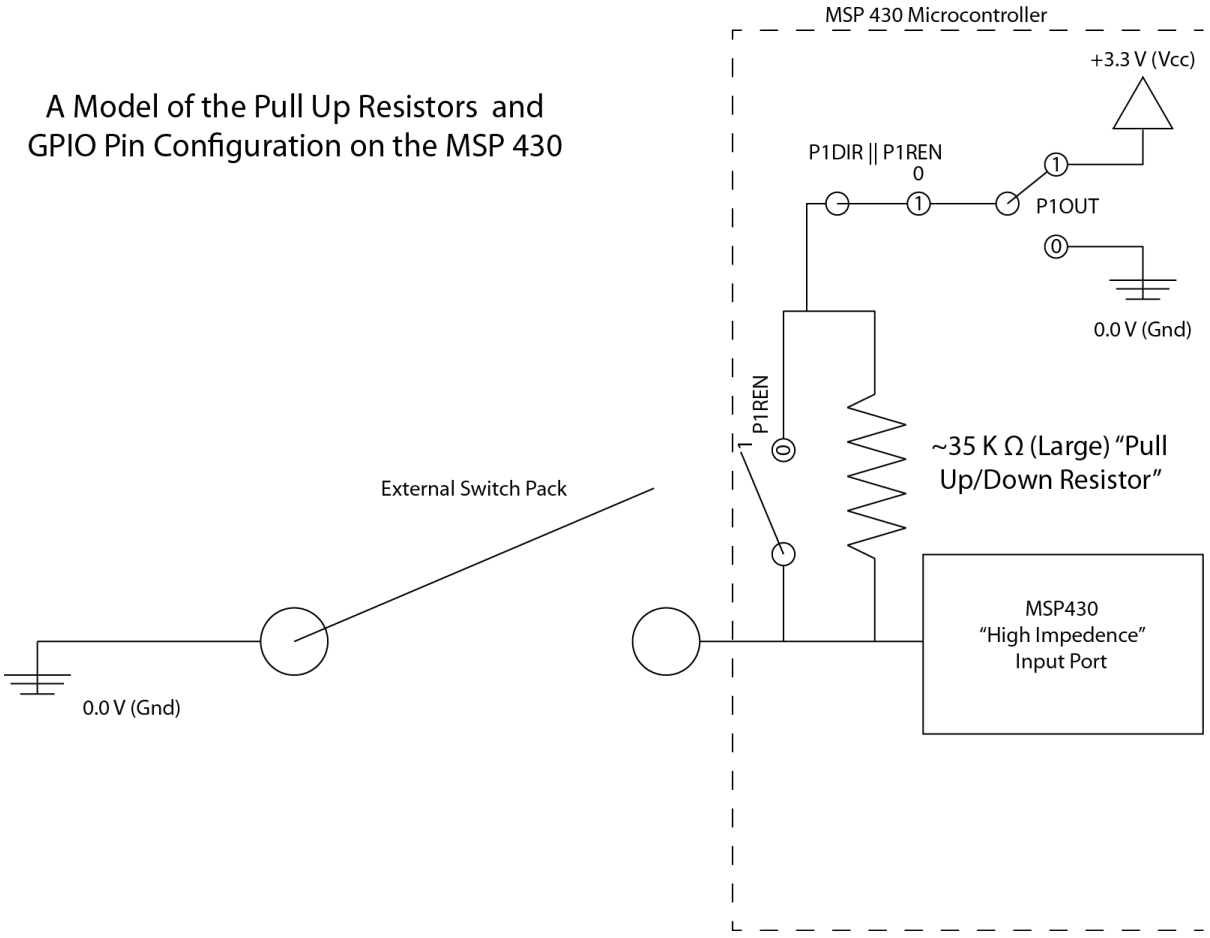
Pull-Ups in the MSP430

For better or for worse, the MSP430 actually has pull up resistors already built into the chip's hardware. Configuring them takes several steps, but once setup they provide all the functionality above without the extra external connections.

- Set the Pin Direction for P1.0-P1.3 to input. (Set bits 0-3 of `&P1DIR` to "0")
- Enable the resistors themselves. (Set bits 0-3 of `&P1REN` to "1")
- Configure the resistors to be pull-up. (Set bits 0-3 of `&P1OUT` to "1")

Note: The most confusing part of the whole process is the double function of `P1OUT`. Because of the hardware implementation on the MSP430, `&P1OUT` controls the outputs as well as the connections to the pull up resistors. **You will need to ensure that every time you output a value, you KEEP the lower four bits "1"**. The easiest way to do this is just by ORing your raw output with the constant `#0Fh` before you write to

`P1OUT`. The MSP430 does not have a specific "or" instruction by name, but `bis` does the same thing. For more info on `bis` and its inverse `bic`, see [next week's lab.]



Notice that configured this way, the MSP430 GPIO pin takes the form of the simplified Pull-Up figure above.

Polling

Philosophy

- A traditional single threaded polling scheme consists of a main loop that runs continuously. Within that loop, the processor periodically checks for changes, and if there are none, continues looping. Once a change is detected, the program moves to a new section of code or calls a new subroutine to deal with the changes.
- Polling has advantages and disadvantages-- it keeps program execution linear and is very easy to code and implement, but it also is not incredibly responsive. Since polling only checks values at certain points in the main run loop, if the loop is long or changes occur quickly, a polling scheme can miss input data. For now though it will suffice.

Assignment Details

Your task is to code a simple input to output echo program for the MSP430. Your program should consist of:

- A setup section that runs once and configures the GPIO pins
- A main loop that runs infinitely
- Code inside your loop to read the state of the GPIO input pins
- A separate section of code to write the changes to the output pins and then return to the main loop

Note:
Masking
You should already know the basics of masking from class, but it becomes very important when dealing with I/O. Since different pins do different things in the same port (P1), you the programmer will have to be careful not to accidentally modify the wrong bits even though your instructions will operate on the entire register.

All images drawn by Matt Johnson, Rice ECE

Lab 4-1 Interrupt Driven Programming in MSP430 Assembly
This module contains the Elec 220 lab 4, which covers basic interrupt usage on the TI MSP430 microcontroller at the assembly language level.

MSP430 Interrupts and Subroutines: Your Tasks

This week you will learn more about the philosophy of interrupt driven programming and specifically how interrupts work on the MSP430. To test out your knowledge, you'll write another simple I/O echo program that builds off the code from the last lab.

1. Coding in MSP430 Assembly, **create an interrupt driven I/O echo program**. The program should read the values of the input pins when pin 4 (P1.3) triggers an interrupt, and then output the read value to the 7 segment display. Details

Background Information

A Few More Instructions

Like you saw in the GPIO Lab, the MSP430 (even though it's a RISC **R**educed **I**nstruction **S**et **C**omputing processor) has a fair number of instructions in addition to those you learned for the LC-3. The extra instructions help programmers simplify code readability and streamline program execution.

You've already seen how the MSP430 uses memory access modifiers and the general purpose `mov` instruction to implement all the functionality of the LC-3's plethora of load and store instructions. Two other very useful MSP430 instructions are `bis` (**Bi**t **S**et) and `bic` (**Bi**t **C**lear). These instructions take an operand with "1"s in the bits you wish to set or clear, and then a destination upon which to do the operation. This comes in handy when you need to modify a few specific configuration bits out of a whole register (like the GIE bit in the SR for interrupts... see below!). The header file has pre-defined masks you can use with `bic` and `bis` to make bit operations much more readable.

Note:
The `bis` and `bic` instructions actually emulate functionality you already had with `and`, `inv`, and `or`.
~~~~~~
`bis op1, op2` corresponds to `or op1, op2`
~~~~~~
`bic op1, op2` corresponds to `inv op1 and op1, op2`

Directives

Assembler and Compiler Directives sound intimidating, but they are nothing more than bits of code intended for the assembler/compiler itself. Directives allow you to specify how the assembler/compiler handles your code and how it all finally comes together into the executable binary file.

The skeleton file has included several directives all along-- `.cdecls C,LIST, "msp430g2231.h"` tells your .asm file to include the c code header aliases from the generic MSP430G2231 configuration file. `.text` tells the assembler to place your program in the main flash memory section, and `.sect "reset"` defines where to start the program after a processor restart.

In this lab, you'll have to use directives to [place your ISR vectors into the vector table.]()

Basic Interrupts

Problems with polling

Continuously polling a pin for input wastes useful CPU cycles and consequently uses more power

The CPU must check the pin often enough to detect a change--when trying to catch a rapidly changing digital signal (a small pulse or transient, etc.), polling may not be sufficient.
In conclusion, polling is easy to understand and implement, but is generally inefficient.

The solution... interrupts

Interrupts use dedicated hardware to detect input changes or hardware events (button pushes, timer intervals, etc...)
When a change is detected, the interrupt logic interrupts the CPU execution.

- The CPU stops what it is doing and calls a special section of code determined beforehand in the interrupt vector table. This section of code is known as the **I**nterrupt **S**ervice **R**outine, or ISR for short.
- Once the interrupt has been serviced and the ISR is complete, the CPU returns to what it was doing before.

The way the main program pauses execution and then branches to a new section of code works in a similar way to the LC3's Traps.

Advantages to Interrupts

Interrupts will catch quickly changing inputs (within reason) that polling might have missed.
The CPU is allowed a level of freedom to multitask without needing to "worry" about explicitly catching input changes. The CPU can do other tasks safely while waiting for an interrupt to fire.

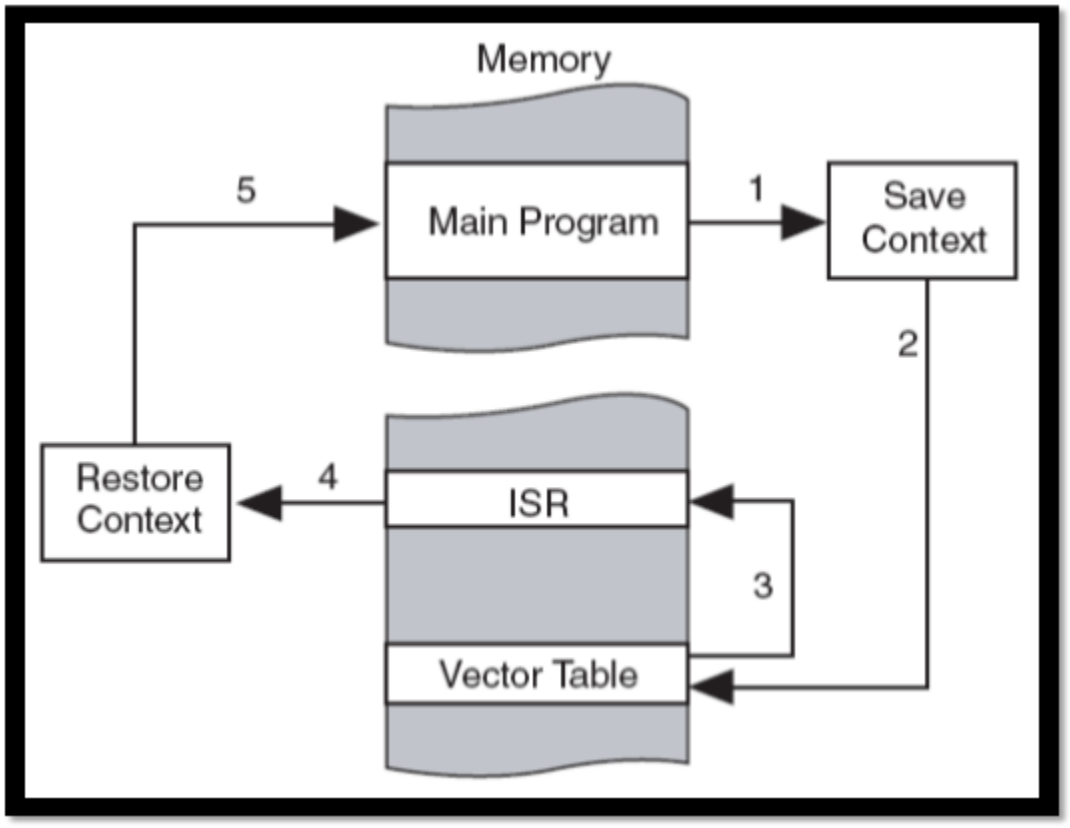
Note: Programs can be "interrupt driven," meaning that the program is just a collection of different interrupt service routines for different tasks.

- The CPU is only active while servicing an ISR, allowing it to go into low power mode between interrupts. Programs that spend a large percentage of their run time waiting on outside events can be made **much more power efficient**.

Basic Interrupt Implementation

Discrete hardware detects interrupt conditions and then triggers the appropriate interrupt in the CPU if it is high enough priority. The interrupt vector table maps each interrupt to the memory address of its interrupt service routine. Like with traps, the CPU first goes to this table to find the address of the ISR and then jumps to the actual ISR code.
CPUs contain several different interrupts to handle different external events uniquely.

MSP430 Interrupt Call Procedure



Interrupts on the MSP430

On the MSP430, there are two types of interrupts: maskable and non-maskable.

Maskable Interrupt

Most interrupts are maskable. Maskable interrupts can be enabled or disabled as a group by setting the GIE (**G**eneral **I**neterrupt **E**nable) bit in the status register. The interrupts must also be enabled individually, but masking allows delicate code (For example, if you are running a precisely timed output routine that must execute all at once) to run in a near interrupt free state by disabling only one bit.

```
Enabling All Maskable Interrupts

bis.w    #GIE, SR
```

Non-Maskable Interrupt

Non-Maskable interrupts will trigger an interrupt at any point in code execution-- they cannot be enabled or disabled on a line by line basis, and they will execute even if the processor is "stuck". Non-maskable interrupts mainly deal with recovering from errors and resets (illegal memory accesses, memory faults, watchdog expiration, or a hardware reset will trigger non-maskable interrupts).

In the MSP430, GPIO interrupt capability must be enabled at the masking level as well as the individual pin enable level. Interrupts should be enabled during the program initialization (before the main code loop or entering low power mode), but after any initialization steps vital to the ISR

There are four main steps to enabling interrupts on the MSP430's GPIO pins.

- Enable interrupts on the individual input pin (in this example pin P1.4) using the port's interrupt enable register. `bis.b #010h, &P1IE` P1IE= **P**ort **One** **I**nterrupt **E**nable
- Select whether the interrupt triggers on a transition from low->high ("0") or high->low ("1") using the port's edge

select register `bis.b #010h, &P1IES` P1IES=**P**ort **One** **I**nterrupt **E**dge **S**elect

- Clear the interrupt flag on the pin in the port's interrupt flag register. `bic.b #010h, &P1IFG` P1IFG=**P**ort **One** **I**nterrupt **F**la**G**

> **Note:**Flags are important. For one, if you forget to clear the flag at the end of your ISR, you will just trigger another interrupt as soon as you return. Also, all of the GPIO pins trigger the same port one ISR. If you have multiple interrupt triggering pins, flags can allow you to determine which pins triggered the interrupt.

- And lastly, only after all of your other important setup, enable all the maskable interrupts in the overall CPU status register. `bis.w #GIE, SR`

Writing an MSP430 Interrupt Service Routine

The ISR needs to be a section of code outside of the normal main loop.
Your ISR must begin with a label and end with a `reti` instruction. `Pin1_ISR <YOUR ISR CODE> bic.b #001h, &P1IFG reti`
At the end of your .asm program, you need to tell the assembler to write the starting address of your ISR to the correct section of the interrupt vector table. The label at the beginning of your ISR allows you to find this address.

> **Note:**CCS4 uses a separate file to define different sections of your controller's memory. This extra layer of abstraction makes it easier to port code between different

microcontrollers, but means that you the programmer can't write directly to a specific memory address in your program. To fill your vector table, you'll need to use the following syntax: `.sect MEMORYSECTION .word DATATOPLACE/LABEL`

The port one interrupt vector for the MSP430 G2231 is defined as 0xFFE4. If you look in the file "Lnk_msp430g2231.cmd" (in the file browser for your lab 4 project), you will see that address 0xFFE4 has been assigned to INT02. In the second half of the linker file, the section .int02 has been assigned to memory addresses > INT02.
When you want to write to the GPIO entry of the interrupt vector table, you need write to code section ".int02" in your assembly file.
`Setting the GPIO vector in the interrupt vector table`

```
.sect ".int02"
.word Pin1_ISR
```

The `.sect` instruction directs the linker to put the code that follows into a specific code section. (You have been using this all along, just putting your code into the main program ".text" section.)
The `.word` instruction directs the linker to write a word length data value into memory.

For more information on interrupts, see the Interrupt section of TI's Microcontroller and Embedded Systems Laboratory.

Subroutines

Subroutine Basics

Subroutines have a lot in common with interrupt service routines (in fact, many programmers use ISR interchangably between **I**nterrupt **S**ub **R**outine and interrupt service routine).
Subroutines are sections of code you use repeatedly during a program-- they allow you to keep repetitive program sizes smaller by re-using the same code section instead of repeating it everywhere you need it.
To go to a subroutine, use the `call #SubroutineLabel` instruction. `call` is analogous to triggering an interrupt. Call works in practice a lot like just jumping to the label, but it also pushes the PC onto the stack (like an ISR) so you can return to wherever you may have left off (since multiple places in code can call the same subroutine).
At the end of your subroutine, use a `ret` (**ret**urn) instruction to pop the PC off the stack and go back to the original execution point of the main program. This is analogous to the `reti` instruction at the end of an ISR.

Note: Calling a subroutine on the MSP430 **ONLY** saves the PC, not the status register like an ISR. You can use subroutines to encapsulate complicated logic, and then examine the conditions afterwords in your main program.

There is a **slight** performance trade off when using subroutines from the overhead involved with storing the PC and moving to a new section in memory, so use them intelligently.
A simple subroutine to demonstrate call and return: `<Your Other Code...> call #Sub220 <Your Other Other Code...> Sub220 add R4, R5 inv R5 ret`

Interrupt Assignment Detail

Your task is to create a simple MSP430 assembly program using CCS4 and the MSP430 LaunchPad to output a stored value to the 7-segment display.

Your program should be **interrupt driven**, and triggering an interrupt on switch 4 (Pin 1.3/LaunchPad Pushbutton S2) should store and output a new output value corresponding to the state of switches 1-3. Changing switches 1-3 should **not effect the output until toggling switch 4**. Your program should consist of:

- A setup section that configures the GPIO pins and enables interrupts.
- An infinite main loop that does nothing (the **N**o **Op**eration instruction `nop` could come in handy).
- An ISR that takes the new inputs and writes them to the output before returning to the main loop.

Interrupt Diagrams Courtesy of TI document slau144e, "MSP430 User's Guide."

Lab 4-2 Putting It All Together
This Lab Module has the project outline for a simple assembly display program. The program is interrupt driven and combines GPIO, polling, and interrupt concepts.

A More Complicated Assembly Program

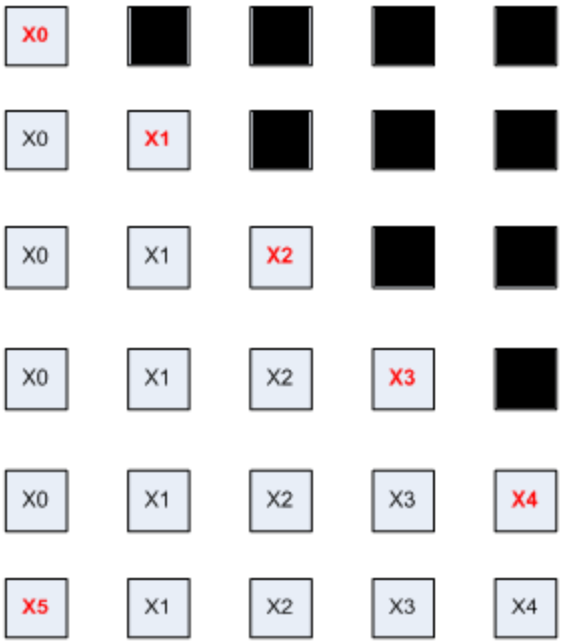
By now you already have all of the tools you need to complete this assignment. Remember what you have learned about MSP430 assembly language, setting up GPIO, and using interrupts.

1. Coding in MSP430 assembly, **implement an interrupt driven number sequence recorder**. You will use the same input configuration from last week (get data from pins 1-3 on an interrupt from pin 4), but now will output a readable loop of the last 5 received numbers in order. Assignment Details

Part II Assignment Detail

Your task is to write an assembly program to display a programmable sequence of 5 numbers on the MSP430 LaunchPad.

> You should use five slots to store the input numbers.
> Since our simple LaunchPad setup only has one display, you will have to rotate through each of the five numbers after a "short" (in human terms) delay.
> Use an ISR to store a new number in the "next" slot. (Next not necessarily meaning what is currently being displayed). The input should go from slot 1 to 2 to 3... etc. regardless of which slot is currently being output.
> The program should only display a slot after a number has been input into it. You will need to keep track of which slots have been filled.



Only grey boxes are output to the display. Also, notice how after filling all five slots, the ISR loops back and starts filling from the beginning.

Your program should consist of:

- A setup routine that readies all the components of your program.
- A main loop that displays the stored numbers one after the other with a readable delay in between.
- An ISR that stores each new input number to the appropriate slot.

A Few Hints:

The MSP430 operates at ~13MHz, which may seem slow in terms of computers, but is much too fast for the human eye to process (~30Hz). You will have to implement a very significant delay in between number changes.

One way to generate a naive delay is a long loop which does nothing. You may even need to use a nested loop depending on how long of a delay you need.

Nested Loop Example in C:

```
int i=0; int j=0; for (i=0; i<bigNumber; i=i+1) { for(j=bigNumber; j>0; j=j-1) { <!--This code will run ixj times-->; } }
```

You may find it convenient to put your five slots in RAM instead of using registers. You can then store a memory address in the register, and then increment it or set it as needed. You will need to use indirect addressing mode though. `mov R4, 0(R15); moves the contents of R4 to the address in R15` `mov 0(R15), R4; moves the contents of the address in R15 into R4` `mov &0x0200, R4; moves the contents of memory address 0x0200 into R4`

Consider where it may be useful to implement parts of your program in subroutines

Wrapup

Congratulations on completing lab 4! Your program sophistication has dramatically increased. You understand the basics of **interrupt driven programming**, and know how to use assembly level **subroutines**. You have had to keep track of **data** as well as design a responsive I/O interface to the outside world. Keep up the good work!

Labs based on the original Elec 220 labs maintained by Michael Wu.

Images from original lab documents by Yang Sun. Modified by Matt Johnson.

Lab 5-1 C Language Programming through the ADC and the MSP430

The C Language and Analog Interfacing: Your Task

This lab covers the basic principals behind Analog to Digital Conversion, as well as the basics of programming in C. You are expected to have some background in C from class, but if you are confused, see this basic reference.

1. Using Code Composer Studio 4, write a C language program turning your MSP430 LaunchPad into a simple 10 level voltmeter. Your program should divide the 0-3.3V input range of the ADC into 10 zones, and then output from a 0 to a 9 on the LED display depending on the input voltage. **DO NOT EXCEED AN INPUT VOLTAGE OF 3.3V**. You will damage your circuits and destroy your MSP430. Assignment Details

The ADC and "C" Through a Practical Example

Interfacing with the Analog World: The ADC

ADC's play an incredibly important role in digital electronics and DSP. ADC stands for **A**nalog to **D**igital **C**onverter, and it does exactly what you would expect it to. It samples an external voltage, and then converts that voltage to a binary number compared to the reference voltage range from Vdd to Vss. (In plain English terms, the ADC samples what fraction the input is of some maximum allowed reference voltage.) The ADC's result gets written to a memory mapped register, where the programmer can access it and use it in his or her code.

An ADC has a **finite voltage range it can safely convert** (usually related to its power supply range, but not always). The precision of the converted sample is related to the number of bits used by the ADC. **More bits means more precision** (more finite "slots" into which the infinitely variable analog single can be quantized) and a lower "quantization error." To learn more about error and ADC, see this except from the Introduction to Electrical

Engineering course notes. ADC's also have a **maximum sampling rate specification** (how frequently the ADC can make a conversion), but in this course we will be sampling very low frequency signals, so we won't need to worry about it.

The MSP430 ADC

The MSP430 G2231 has one 12 channel 10 bit 200Khz ADC. ADC channels allow the single ADC to select between several different signals (such as two different analog inputs on different GPIO pins) like an analog multiplexer. In the G2231, channels 1-8 are connected to the 8 P1 GPIO pins, and channel 10 is connected to the chip's internal temperature sensor. You can select which channel to convert by setting the `ADC10CTL1` register's (**10** bit **ADC** **Con**t**rol 1**) `INCH` property (**In**put **Ch**annel).

For this lab, we will configure the ADC to use the internal 3.3 Vdd as the reference voltage.

- A voltage of 3.3V would result in the ADC register holding 11 1111 1111 (0x03FF)
- A voltage of 0.0V would result in the ADC register reading 00 0000 0000 (0x0000)
- A voltage of 1.65V would result in the ADC register reading 01 1111 1111 (0x01FF)
- The ADC will have a sample resolution of 3.3V/1024 [Voltage Range/$2^{\#Bits}$], or .0032 Volts.

The ADC is a peripheral device, so it operates independently from the CPU. It has several operation modes (configured by writing to its control registers).

Peripheral
: A device that can operate independently from direct CPU support and control. Peripherals often rely on interrupts to notify the CPU when they have completed some given task or need attention, and use independent control registers for configuration. The ADC 10 is a

peripheral, as well as the MSP430's UART (serial controller) and timers.

ADC10 Operation Modes

- Single Sample and Hold-- the ADC10 will start a conversion when triggered by the CPU. After that conversion, it will hold the converted value in the ADC10MEM register and automatically go into sleep mode until signaled to begin another conversion. We will mostly use this mode.
- Sequence of Channels Sample and Hold-- the ADC10 will convert a series of different channels once, and store the result to ADC10MEM.
- Repeat Single Channel Mode-- it will continuously sample on channel, continuously updating the ADC10MEM register.
- Repeat Sequence of Channels Mode-- the ADC will continuously sample through a series of channels.

The MSP430's ADC 10 also has a built in memory controller. We won't be using it, but it becomes important when using the repeat modes. The memory controller can automatically write the ADC data into main memory as conversions finish, bypassing the CPU.

The G2231's ADC can run off of one of several available clock signals of varying speeds. The ADC10 also has a clock divider that can further slow the conversion speed by up to a factor of 8. Once a sample has been captured, it is held ready in the ADC10MEM register for a defined number of clock cycles. Since we are concerned with a low frequency signal, we will want to slow down the ADC10 as much as possible. (Students who have had Elec241 will notice some fundamental flaws in the assumptions made regarding high-frequency noise, but in practice this has very little effect on the final results). Even in its slowest mode, the ADC10 will still sample too quickly, so the use of some kind of moving average will help stabilize its DC readings.

Controlling the ADC10 in C

C Basics

Your C program will be structured similarly to its assembly language counterparts, but with a much different syntax. Like before, the register names are all pre-defined in the `"msp430x20x2.h"` header file. To set a register, now just use an equals sign and set it like any other variable. For example, you will want to disable the watchdog timer in the first line of your program. `WDTCTL=WDTPW+WDTHOLD;` The compiler will execute the `void main(void)` function first. From that function, you can call any other functions or run any loops that you would like.

C Skeleton Program

```
#include "msp430x20x2.h" //Global Variable Declarations //Global Function Declarations void main(void) { WDTCTL = WDTPW + WDTHOLD; // Stop WDT //Other Setup //Your Program Here //Can call other helper functions, loops, etc. }
```

Configuring the ADC10

The ADC10 has two main control registers `ADC10CTL0` and `ADC10CTL1`, and two analog input enable registers `ADC10AE0` and `ADC10AE1` (**10**bit **ADC** **A**nalog **E**nable **0**/**1**). These registers control all the timing and conversion aspects of the ADC.

ADC10CTL0

ADC10CTL0, ADC10 Control Register 0

15	14	13	12	11	10	9	8
SREFx			ADC10SHTx		ADC10SR	REFOUT	REFBURST
rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)

7	6	5	4	3	2	1	0
MSC	REF2_5V	REFON	ADC10ON	ADC10IE	ADC10IFG	ENC	ADC10SC
rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)

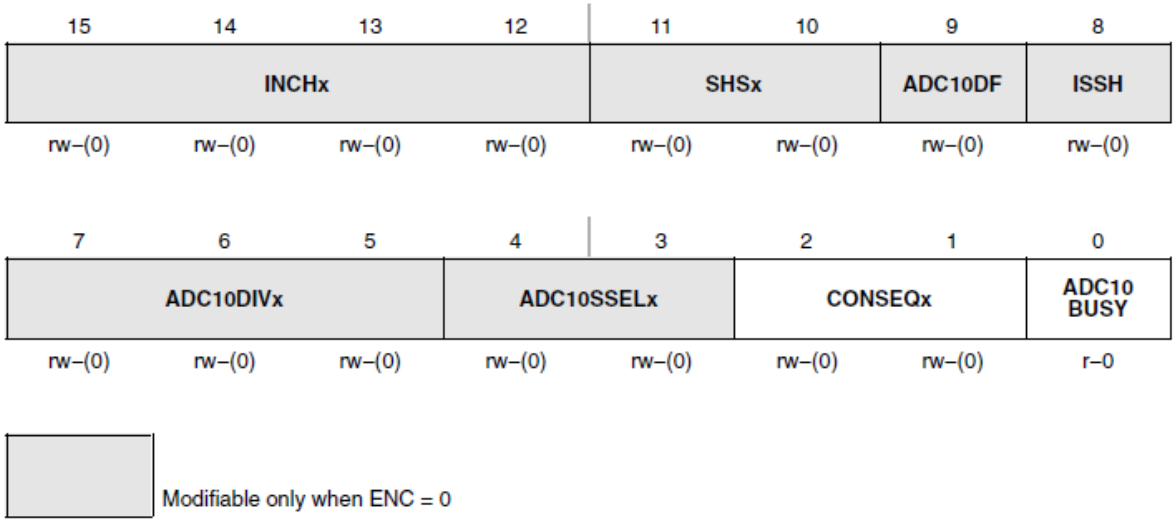
Modifiable only when ENC = 0

In the first control register (`ADC10CTL0`), we only need to change two parameters,

- ADC10SHTx--**10**bit **ADC** **S**ample **H**old **T**ime-- a higher value means each sample will be held for a longer period of time. We want to set this at the max value of `ADC10SHT_3` .
- ADC10ON--**10**bit **ADC** **ON**/OFF--setting this bit to "1" (denoted by the label `ADC10ON`) turns on the ADC, a vital step to performing any conversion!

To actually do this in C, just use addition and an equals sign: `ADC10CTL0 = ADC10SHT_3 + ADC10ON ;`
ADC10CTL1

ADC10CTL1, ADC10 Control Register 1



In the second control register (`ADC10CTL1`), we want to again set two parameters, but we will need to use 4 alias labels instead of just two.

- ADC10DIVx--**10**bit **ADC** clock **Div**ider bit **x**-- for "more flexibility", you set each bit individually in the three bit ADC10DIVx section of the register. Since we want the maximum divider, we will set all the bits.

> **Note:**Since some of the bit labeling is inconsistent (ADC10DIV is bit-wise while ADC10SHT is not), it is always good to examine the header file for a controller to see how its aliases are defined before using them in your code.

- INCHx--**In**put **Ch**annel #-- this 4 bit section determines which of the possible input channels the ADC will actually convert in single convert mode. In series mode, this determines the highest channel to be converted in the series (all channels below this number will also be converted).

```
ADC10CTL1 = ADC10DIV0 + ADC10DIV1 + ADC10DIV2 + INCH_X;
```

Lastly, the ADC10 has the ADC10AE0/1 registers that enable analog input on the different pins. These act as gates to prevent leakage current from flowing from a pin set as an output through the ADC to ground-- a substantial waste of power. To enable the ADC for your desired GPIO pin, just set the corresponding bit in `ADC10AE0` to "1". `ADC10AE0 |= BIT#;`

For more info about the ADC10's configuration options, see the MSP430 manual starting on page 609.

Using the ADC

To read a sample from the ADC, just read from the ADC10MEM register after the sample has completed. `my_var= ADC10MEM;` Remember that we have setup the ADC for single conversion and hold, so if you want another value, you will have to tell it to sample and convert again. You do so by modifying two values in ADC10CTL0:

- ENC--**En**able **C**onversion-- locks in the ADC settings and stabilizes it for conversion.
- ADC10SC--**10**bit **ADC** **S**tart **C**onversion-- setting this bit to one actually triggers the ADC's conversion cycle.

```
ADC10CTL0 |= ENC + ADC10SC;
```

Note: Be sure to use OR equal (|=) so that the configuration bits you set before don't get overridden.

Note: Also, don't forget to configure P1 as usual. You will need to set the pins you wish to use as ADC inputs to input mode at the `P1DIR` register as well as the `ADC10AE0` register. You can configure the P1 registers using aliases and variable assignments just like with the ADC registers.

Assignment Details

Using Code Composer Studio 4, write a C language program turning your MSP430 LaunchPad into a simple 10 level voltmeter. Your program should divide the 0-3.3V input range of the ADC into 10 zones, and then output from a 0 to a 9 on the LED display depending on the input voltage. Don't worry about a value landing on the boundary between two zones, just deal with it consistently. Test your volt meter by attaching it to some of the variable power supplies around the room. **DO NOT EXCEED AN INPUT VOLTAGE OF 3.3V**. You will damage your circuits and destroy your MSP430.

Your Program should consist of:

- A "void main(void)" function that drives your program
- A successful setup routine that properly configures the ADC10
- An output routine that successfully re-scales the 1024 ADC possibilities to 10 zones

Lab 5-2 Using C and the ADC for "Real World" Applications with the MSP430
This lab assignment explains the principles of calibration, and then lays out an assignment for the last in class Elec 220 lab.

A Real World Situation

On some level, every signal and every interface starts out analog. In this lab you will learn a simple two point calibration routine and how to use it to get accurate data from the physical world. This assignment is less about new programming principals and more about applying what you already know. You have one main task:

1. Using Code Composer Studio 4, write a C language program to drive a precise 3 digit volt meter. Use the ADC to read the voltage and display back the actual voltage value. Write an interrupt driven calibration routine for your volt meter and an output routine that will allow you to display the three output digits. Assignment Details

Analog Signals Background

Simple Analog Sensing

The analog voltmeter may seem simple, but the ability to measure and quantify an analog voltage allows the MSP430 to interface with a whole range of analog sensors. Ultimately, any real world signal starts out analog, so at the heart of every interface lies some kind of analog system..

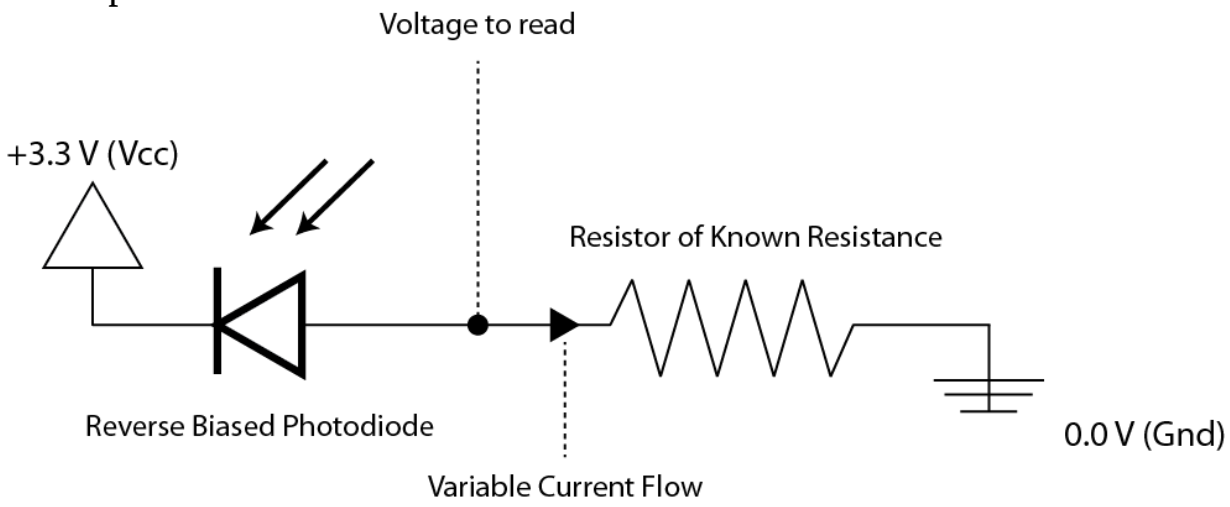
Analog sensors use the physical properties of some electronic device (or a system of many devices) to modify an analog voltage or current signal. The MSP430's ADC allows it to use that signal in computations (as long as the signal's maximum frequency is less than 100khz- the nyquist sampling frequency of the 200khz ADC). In simple terms, the ADC can't accurately measure signals that change too quickly for it to see. Not only that, but the ADC can pickup unwanted distortion from those higher frequency signal components, making even the low frequency parts inaccurate.

Example:
A simple analog device
A **photodiode** is just one of many such devices. When kept in reverse voltage bias, the photodiode allows an amount of current through it proportional to the amount of light shining onto it. By attaching a photodiode in series with a resistor, we can examine the voltage across the resistor to find that current (v=i*R), and therefore the relative amount of light!

Photodiode

> A P-N diode specifically constructed to allow a large amount of light to enter the diode's depletion region. This excess light generates free electron and hole pairs in the depletion region, allowing current to flow. Photodiodes are surprisingly linear, meaning the light flux is almost 1:1 proportional to the amount of current output. To learn more about photodiodes and electronic devices in general, look forward to ELEC 305.

A Simple Photodiode Circuit



Our Simple Circuit

For simplicity, we're going to **continue using the voltage sources in the lab to simulate an analog device.** As you continue to take more courses and learn more about analog electronics, you will be able to design your own analog circuits to capture and condition the information you want.

In this last lab you'll be using the full repertoire of I/O options available to you. You'll use the ADC to read an analog voltage, the pushbuttons and interrupts to control a calibration routine, and the 7-segment display to output one digit of a measured number at a time.

Calibration

Like anything in the real world, your sensors and devices won't necessarily work perfectly all the time. Many analog semiconductor devices are light, temperature, and pressure sensitive, so your actual readings can vary depending on outside conditions. Also, while sensors are usually manufactured to within pretty tight tolerances, every sensor has some finite error that you will need to account for to get maximum performance.

Some kind of calibration routine can help alleviate many of these concerns. By calibrating your sensors against a trusted source, you can correct a lot of the skew caused by outside conditions. Calibration will also eliminate any steady state error caused by the device itself.

A calibration routine collects a set of known data points and scales the input signal to fit the calibration data. Since we are calibrating a linear ADC, we only need two data points to determine its operating curve. To get the most out of the calibration, you should use two data points that are as far from each other as possible. **In the case of the MSP430 G2231, that is 0v and 3.3v**.

Calibration Routine Pseudocode (NOT IN C SYNTAX)

```
flag=0;
interrupt function: calibrate()
returns:  lowMeasured, highMeasured
{
        if flag=0:
        set low and set flag=1
```

```
        if flag=1:
        set high

        return
}
```

Interrupts in C

Code Composer Studio has a special way of handling interrupts in C. It uses "compiler directives" (special instructions to the compiler, assembler, and linker) to specify which functions should go in the interrupt vector table. ISRs in C are written like any other function, except that they can take and return no values. This fits with the convention that ISRs don't interfere with or depend on the code around them.

Formatting an Interrupt in Code Composer C

```
#pragma vector=PORT1_VECTOR
     //compiler directive saying that this function should
correspond with the port1 interrupt vector
__interrupt void interruptHandle()
{
        //your ISR CODE
}


void main()
{
... all setup

__enable_interrupt();
        //Enables general maskable interrupts

... the rest of your program
}
```

Other Concerns

Even though interrupts should work in isolation, you often need to get or modify data inside them. This can be done using volatile global variables. A global variable simply means a variable that any function in your code can

see and modify. Using global variables is generally discouraged (they can be easily abused and lead to problems if you repeat common variable names), but in this case they allow you to interface with your ISR. Volatile is a directive that tells the processor to never cache the variable's value. When dealing with ISRs, cached variables could lead to problems if the ISR updates the cached data while another function is using it. Be careful using the volatile keyword excessively though-- the lack of caching slows down performance.

Full Range Voltmeter Assignment Details

Using Code Composer Studio 4, write a C language program to drive a precise 3 digit volt meter. Use the ADC to read the voltage and display back the actual voltage value. Write a calibration routine for your volt meter and an output routine that will allow you to display each of the output digits.
Your program should include:

- A setup section to setup the GPIO, configure the ADC, and enable interrupts
- An ISR that runs the calibration routine and keeps track of what has been calibrated so far (so it will only run once)
- A main loop that continuously samples, filters, and scales the ADC input to the 0-3.3 volt range as determined by your calibration.

MSP430 LaunchPad Test Circuit Breadboarding Instructions
This module contains step by step instructions and pictures showing how to assemble the MSP430 breadboard kit used in Elec 220 for very simple I/O. It uses the TI MSP430 Launchpad development board, a double breadboard, and some basic circuit components (switches and a 7-segment display) to establish an easy to understand I/O scheme.

Breadboard Basics

The solder-less breadboard is a convenient way to setup simple circuits and make connections quickly between electronic components. Each hole in the breadboard has a spring clip that makes a connection to the wire/ IC lead you put into it. **The breadboard connects each vertical row of holes in the main secion**, giving you five holes where you can tie together parts of an electrical circuit. Any time there is a gap between two adjacent holes, they are not automatically connected together.

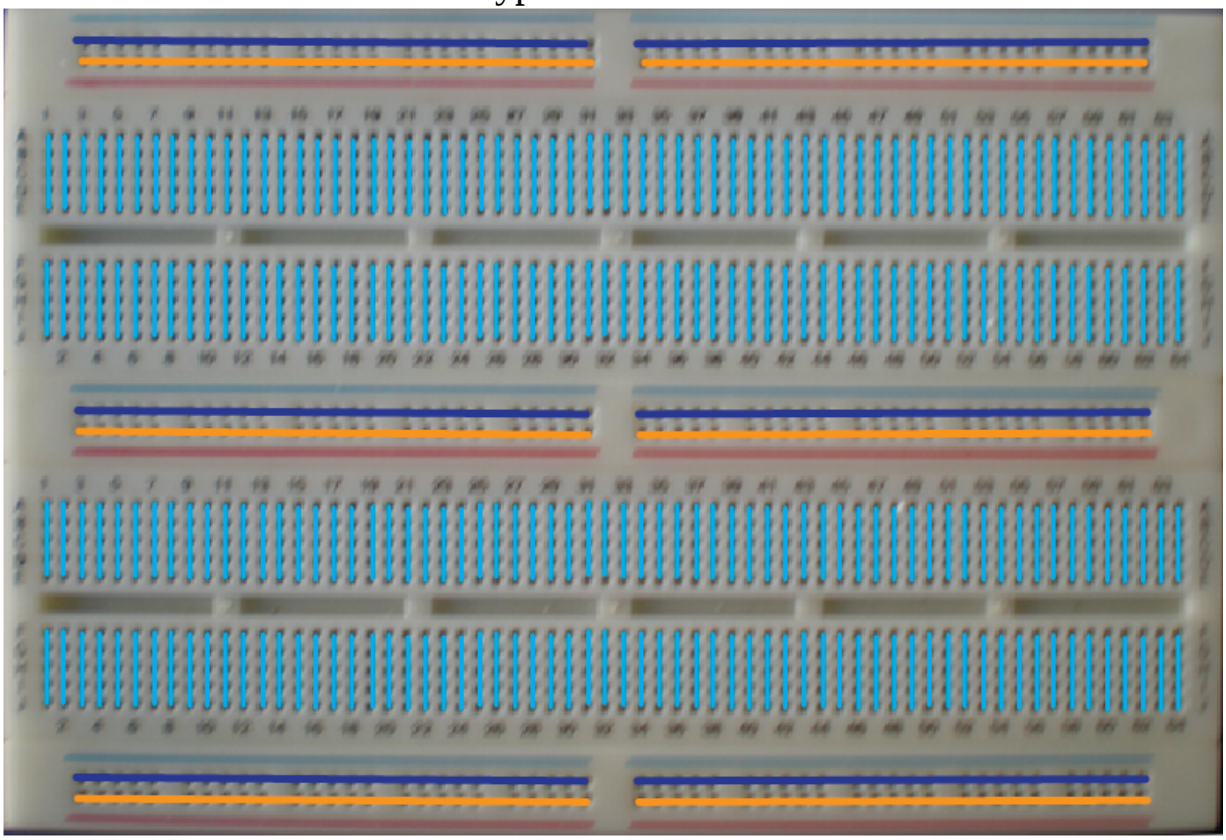
You will notice some red and blue horizontal lines of holes in the top, bottom, and middle of the breadboard. These are your **busses**. In most simple breadboards you will use these for power (VCC and GND) as we do here. The bus sections are not automatically connected together. If you want power to all of the busses, you need to connect them all together as shown below.

Lastly, notice the divided **channel** down the middle of the breadboard. This channel is specifically sized for DIP packaged integrated circuits. You can put a chip across the channel, and have access to each of its pins using the vertical rows above and below. Always put chips across the channel, otherwise you will connect the opposite side pins together and your circuits won't work as expected.

Note:
Be careful when removing chips
If you use your bare hands to try and remove a chip from the breadboard (you will notice it can be difficult, especially with components with lots of

pins), there is a good chance you will end up with the chip plugged into your finger afterwords. You hand has a natural tendency to rotate things when you are pulling hard, so watch out. **There are some IC removing tools around the lab, use one of them or ask your labbies to help you if you accidentally misplace a chip.**

The Connection Scheme in a Typical Breadboard



An illustration of how the breadboard holes are connected together in the breadboards used for the lab. Notice that no connections cross the center channel and that the busses are connected horizontally while the main face of the board is vertical.

Steps for Assembling the Breadboard for Elec 220

Note: Make all connections with the MSP430 disconnected from USB power.

Note:
Be careful when removing chips
See above for explanation. **Use an IC removal tool or ask your labbies for help.**

Tinkerers:
The wiring below is suggested and works well, but if you have any ideas on how to improve the circuit, feel free to implement them in your breadboard. Just be sure that you can successfully run the test program at the end.

1) Get Your Materials

You will need:

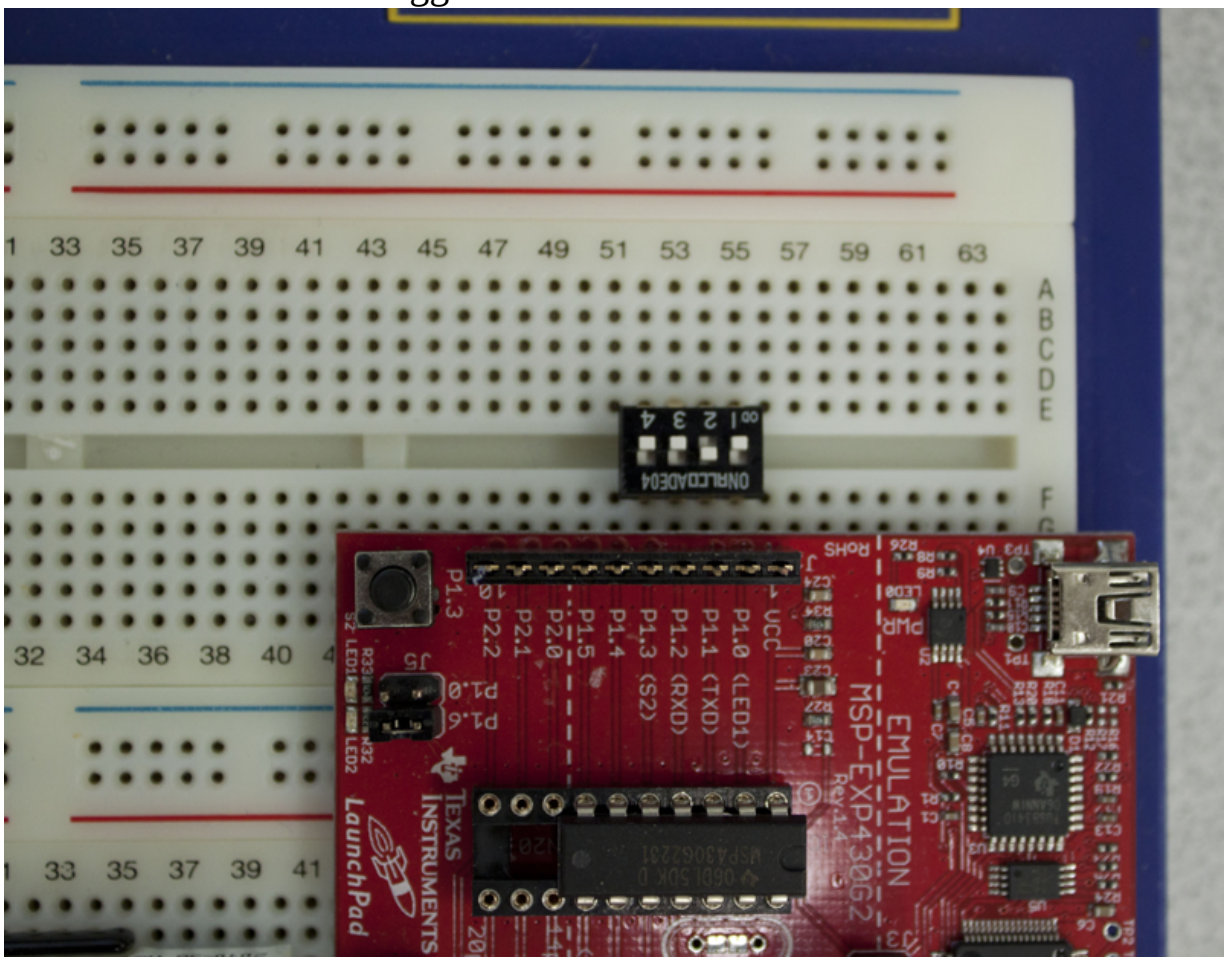
- 1 MSP430 Launchpad
- 1 Double Width Breadboard
- 1 Breadboard Wire Kit
- 1 MC14511B Binary to Decimal Converter
- 1 Kingbright Green 7-segment display
- 1 4 Switch SPST DIP Switch Pack
- 2 470 ohm isolated resistor arrays

2) Attach the Circuit Components to the Breadboard

Most of the breadboards should already have their components pre-inserted in the correct places by the labbies. This section is included for completeness and so you can double check their work if necessary.

The MSP430 launchpad goes in the middle right hand side of the board. **Its upperleftmost pin goes in hole H47**, and the chip sits across both halves of the breadboard. Once it's in, put in the 4 toggle dip switch across the channel above the Launchpad. **The switch's top leftmost pin goes into hole E52.**

The MSP430 and DIP Toggle Switches



Connecting the switch to GPIO pins 0-3

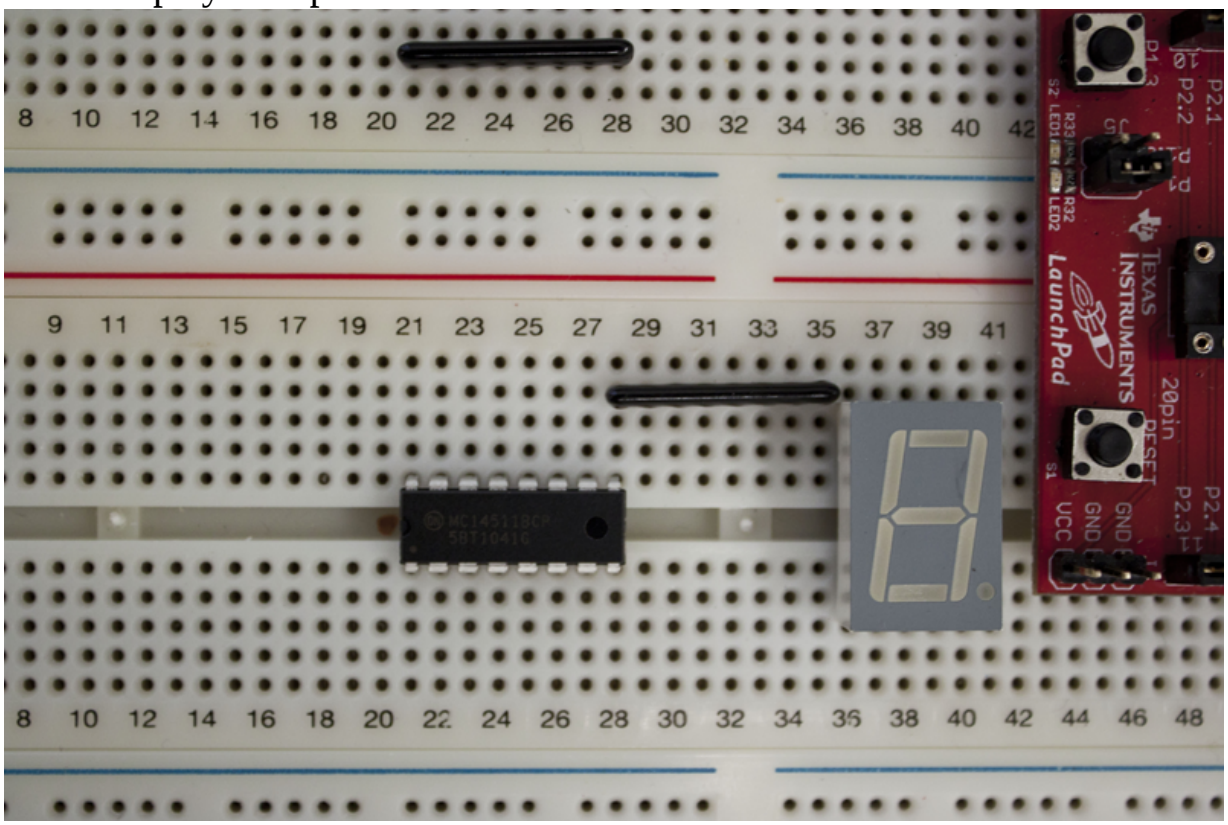
Now insert the decoder IC (the 16 pin chip MC14511BCP) into the board. The **top leftmost pin goes into hole E21** across the lower breadboard channel.

Then put the 7-segment display across the same channel, with its **top leftmost pin in hole C36**.

Take one of your resistor arrays (the long black sticks with a rubberized coating) and place it with the **leftmost pin in hole B28**, connecting it to the top right pin of the decoder by doing so.

Lastly, put the other resistor array with its **leftmost pin in hole I21 of the upper half of the breadboard.**

Main Display Components



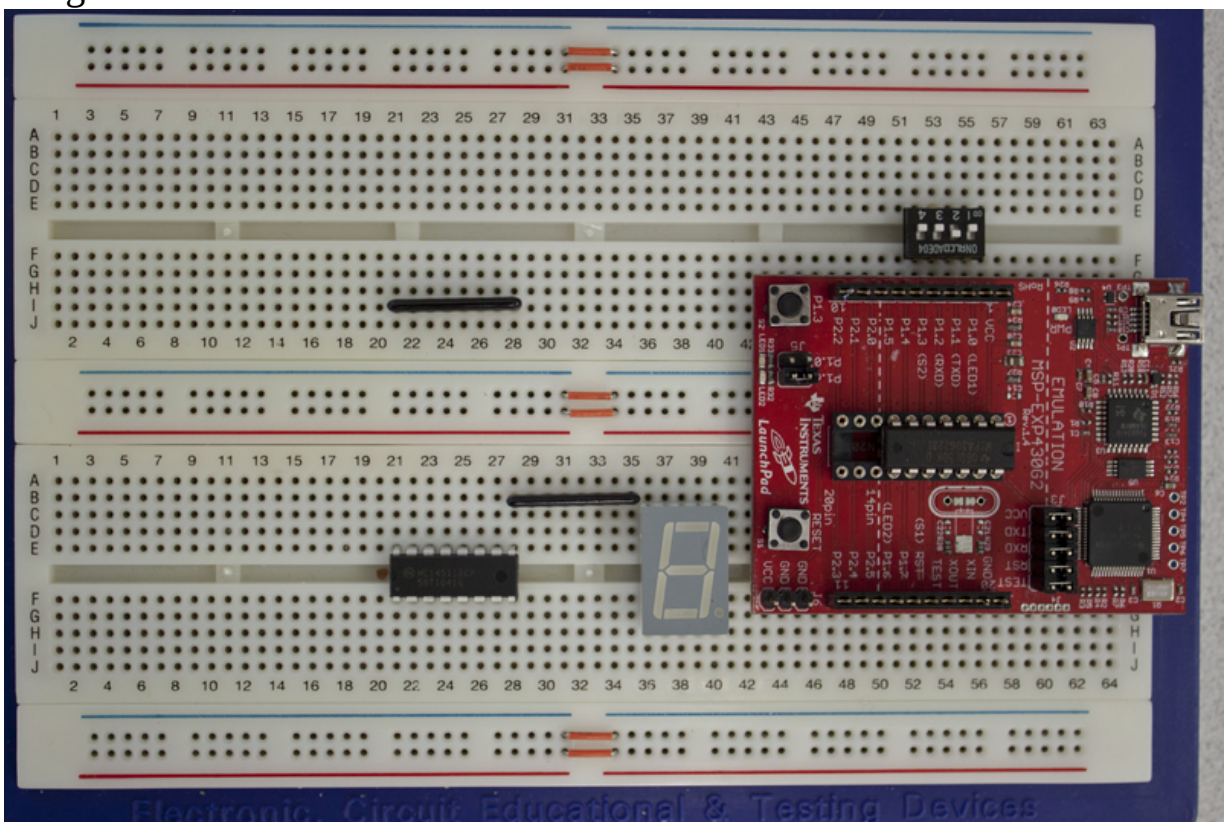
The components are close together to prevent long wire runs and simplify the connections between them (taking advantage of the breadboard connections whenever possible).

3) Wire Your Power Busses Together

To give some additional flexibility, the breadboard busses aren't automatically connected together. In our applications, we will want to run all circuits off of the same power used by the MSP430 itself, so we need to tie all the different bus sections together.

Using the orange wires in the wiring kit, connect each half of all the horizontal bus strips. The wide gaps in the middle of the strips indicate that there is a gap we need to bridge with an external jumper wire.

Lengthwise Bus Connections

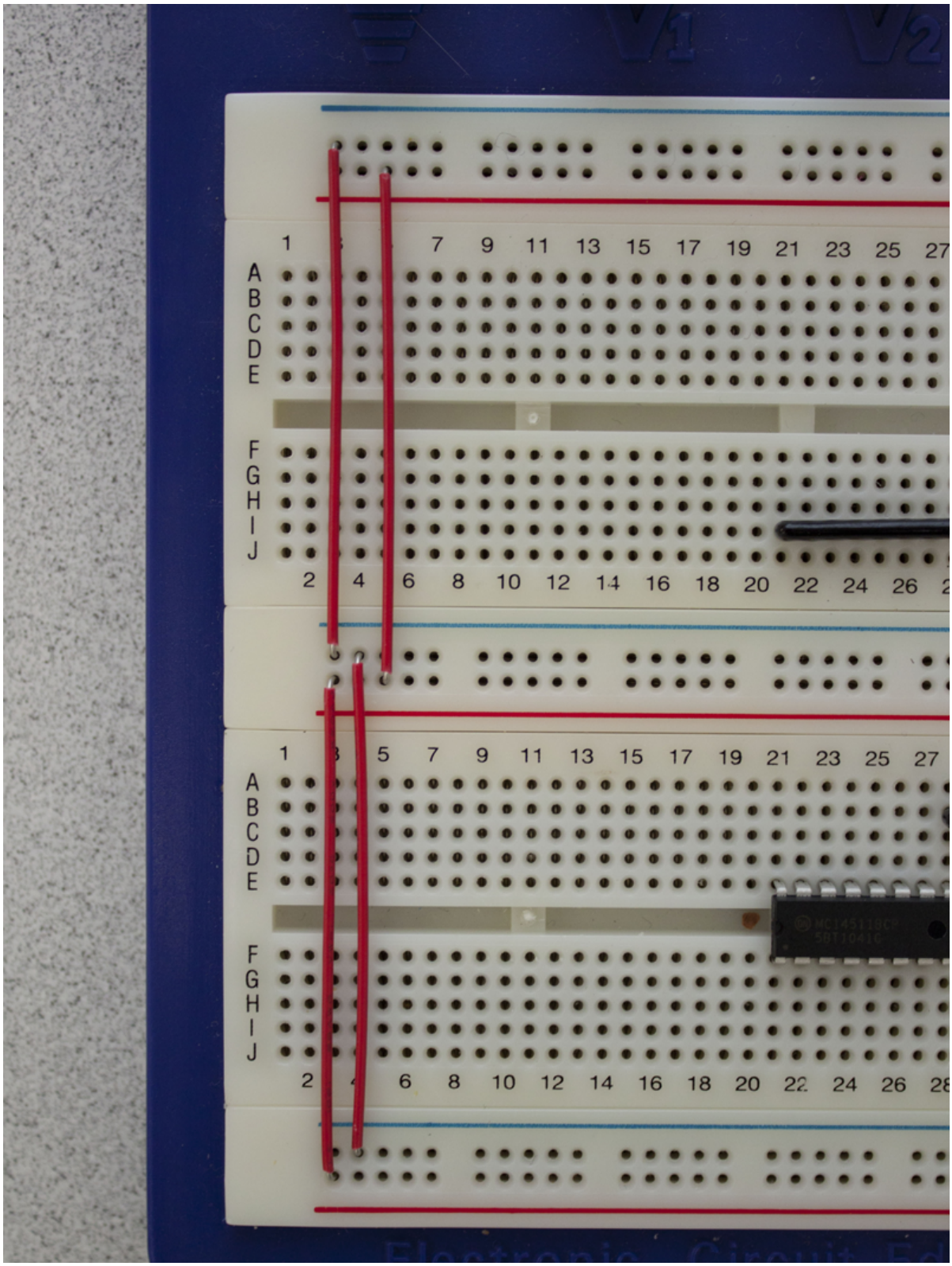


Notice how the orange and red wires in your kits are sized to exactly bridge the respective gaps in the breadboard.

Since the lab setup only needs one power level, for convenience tie all the power busses together. This will allow shorter runs from the chips to whichever bus is closest. **Using the red wires, connect all three red and**

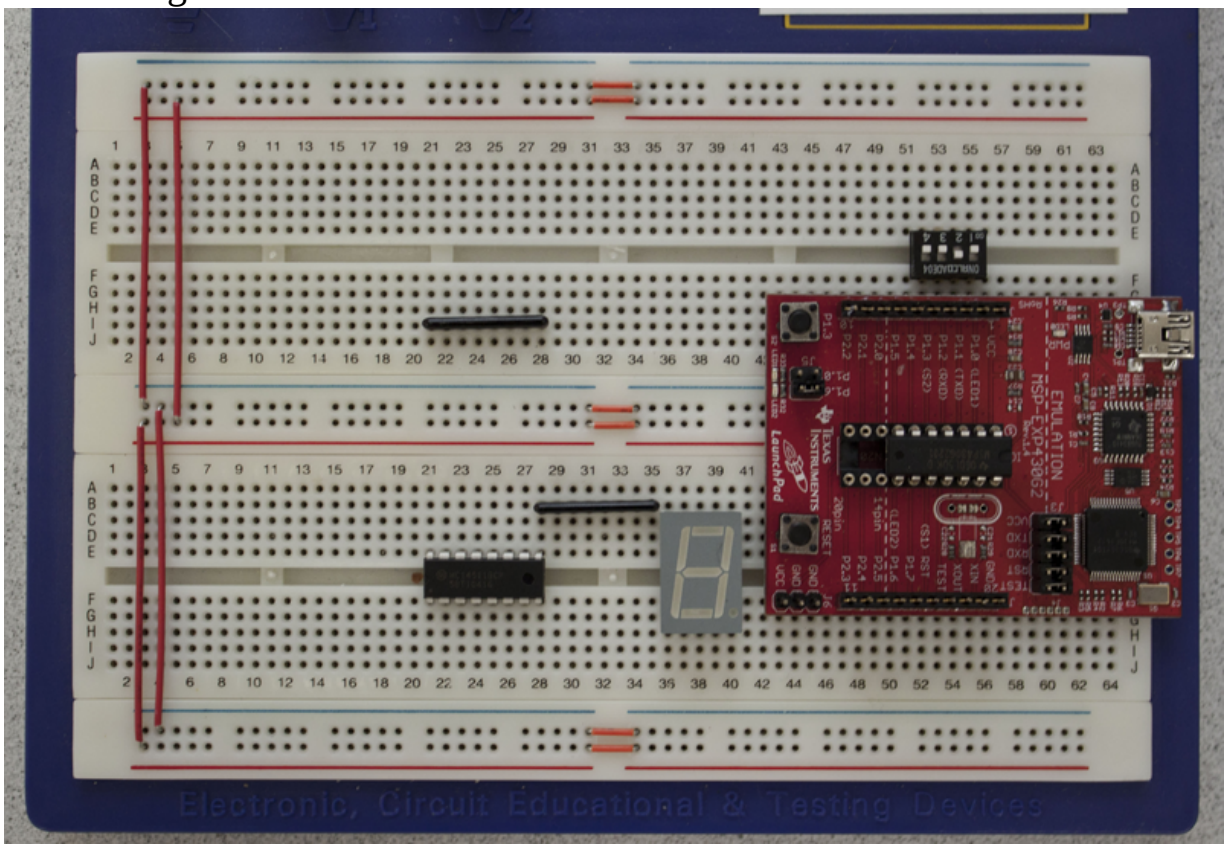
all three blue busses together. Put the connections off to the far left hand side of the board out of the way of your main circuit.

Vertical Bus Connections



Notice how the orange and red wires in your kits are sized to exactly bridge the respective gaps in the breadboard.

Bus Wiring Overall

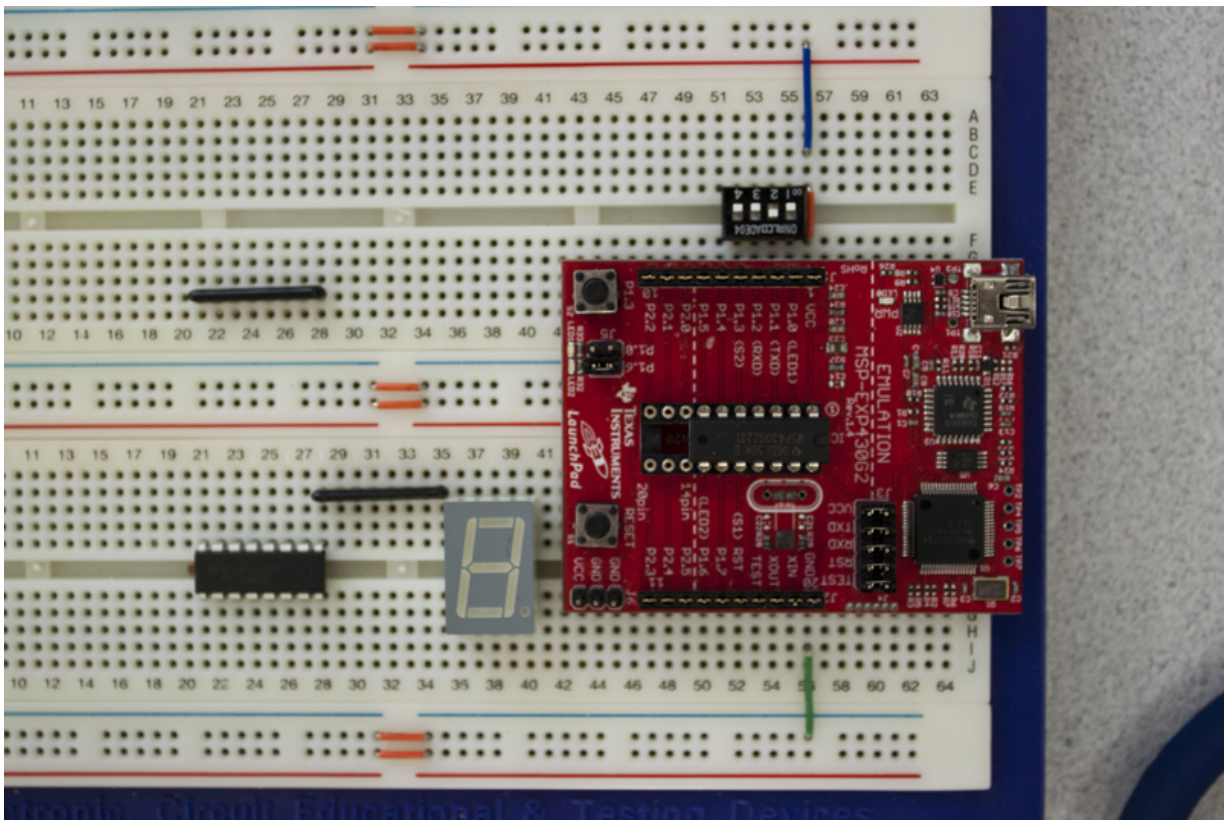


Check your work once you are done-- now there should be one continuous connection between all of the red busses and a separate one between all of the blue busses.

4) Connect Power to Busses

We already wired the busses together, but now we need to connect the +3.3v and GND provided by the MSP430 Launchpad's USB connection and voltage regulators. Connect the **blue bus strip to GND (lower F-J column 56) using a green wire** and **the red bus strip to Vcc (upper F-J column 56) using a blue wire.** You will need to use a **small orange jumper** to cross the upper channel as shown in the picture below.

USB Power to Busses



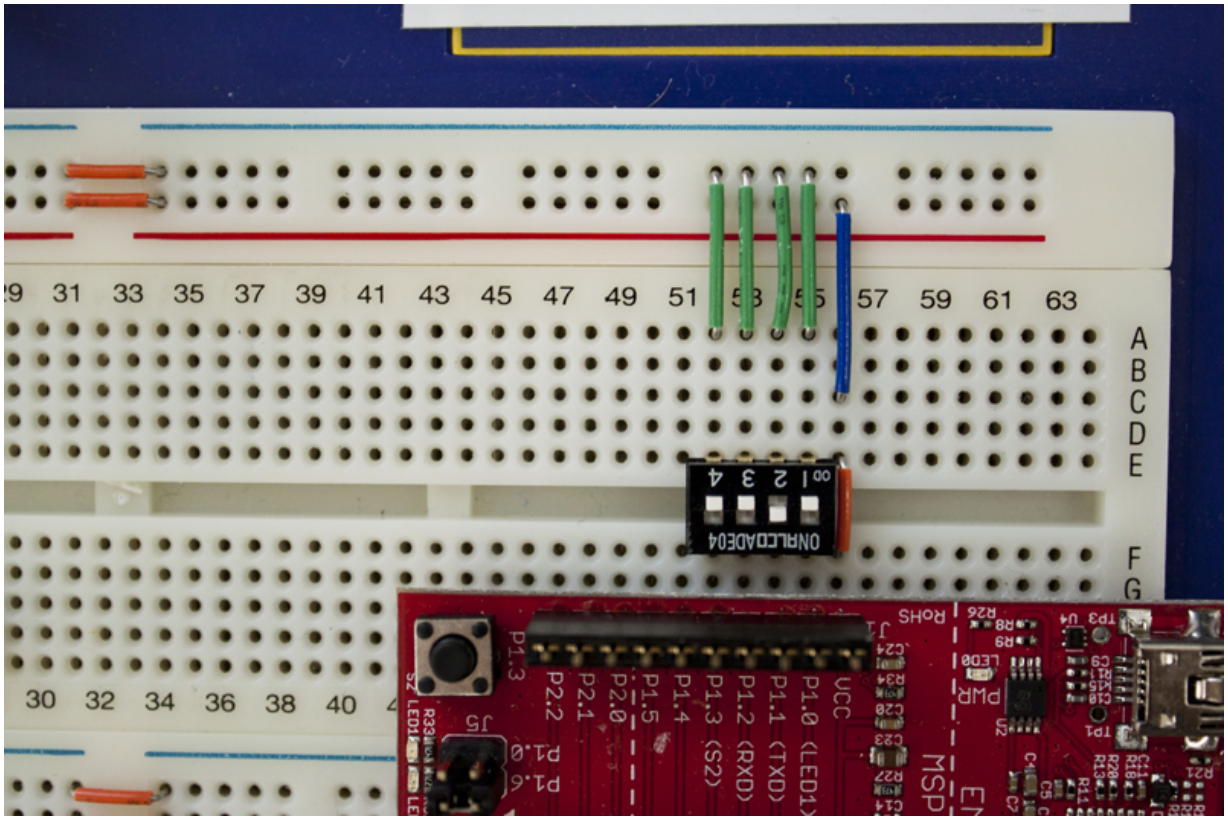
Notice the color choice of wires. For the rest of the construction process, the short green and blue wires are only used for connections to GND and Vcc respectively.

5) Connect Your Switches to GND

Use the **green jumper wires to connect the other side of your switches to GND** (the blue bus). Yes, **the switch pack is upside down**. This is an

intentional design decision that has to do with the pull-up configuration of the MSP430 Launchpad Hardware. With the switch pack upside down, flipping the switch "up" will register a "1" and flipping them "down" will register a "0" like you would expect. See the explanation below for more information, or just take it on faith that it works this way.

Switches to GND



Note:

You may notice that turning the switch on connects it to GND, but turning it off connects it to nothing! This can be really bad in a circuit- the values read from the GPIO pins will be essentially random! Ideally, you would want your switches to be "1" when up and "0" when down. To accomplish this, you can either use more expensive dual pole switches that switch between two connections instead of closing or breaking just one, or use what's called a pull up (or pull down) resistor. This is a resistor of large resistance connected to the rail you want the switch to read when it is open. The GPIO sees most of the connected voltage when the switch is on (and

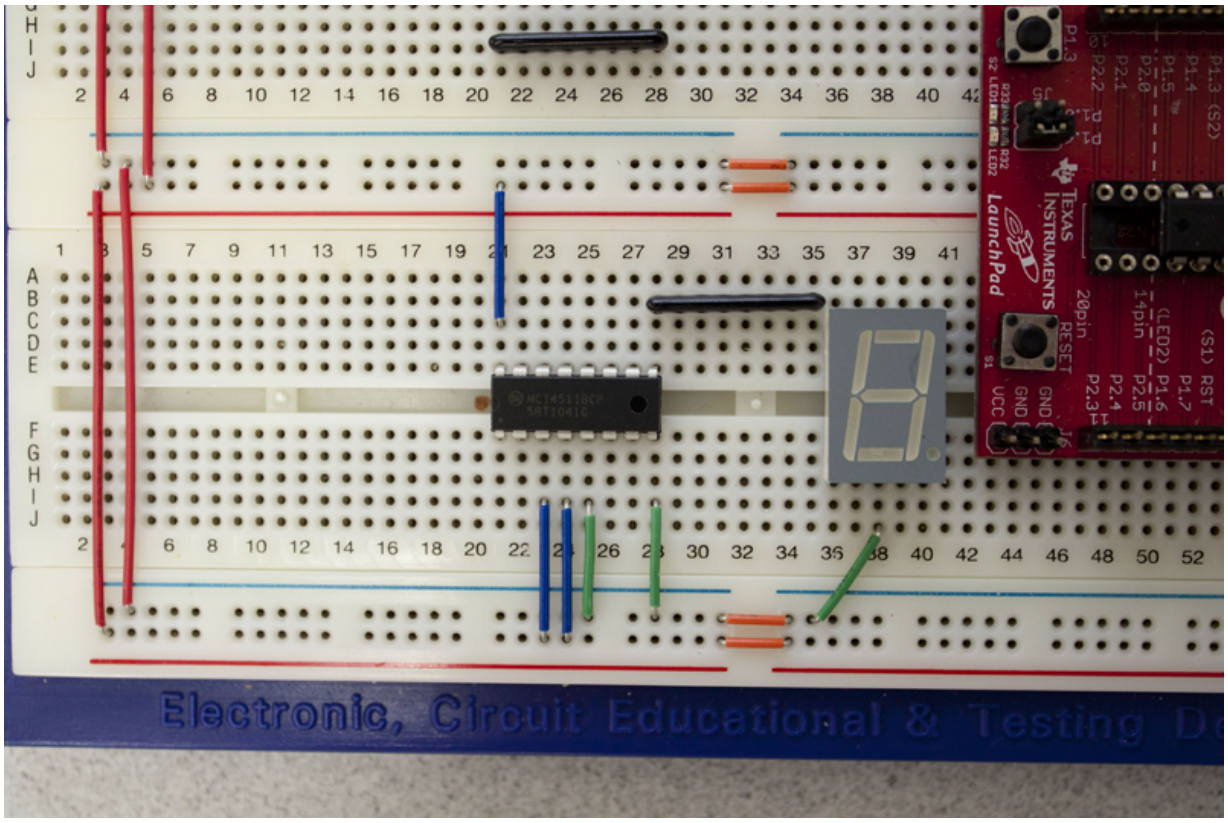
in digital applications most is enough), but sees the other rail when the switch is open.
"But I see no resistors in the picture"-- you're right! The MSP430 has pull up resistors built in- we just have to enable them when we configure the GPIO pins. You'll learn more about that in part II of the lab.

6) Connect Vcc and Ground to ICs

The BCD decoder is an active piece of circuitry, so it needs a power connection to work properly. Connect **Vcc to pin 16 (lower A-E column 21) and GND to pin 8 (lower F-J column 28)**.

The display also needs a common connection to ground (since it is common cathode type). Connect **pin 3 (lower F-J column 38) to GND.** The display is just a package of individual LED's in parallel connected to one common ground point.

Lastly, there are some options (dealing with latching and enabling) on the decoder we want to permanently set in our circuit. **Connect Vcc to pins 3 and 4 (lower F-J columns 23 and 24) and GND to pin 5 (lower F-J column 25).**
Power Connections for Circuit Components



Notice how the color scheme of blue jumpers for Vcc and green jumpers for GND continues here. It's good practice to use a consistent color scheme for Vcc and GND since they can fry your chips if mis-applied.

7) Connect the Decoder to the Resistor Arrays

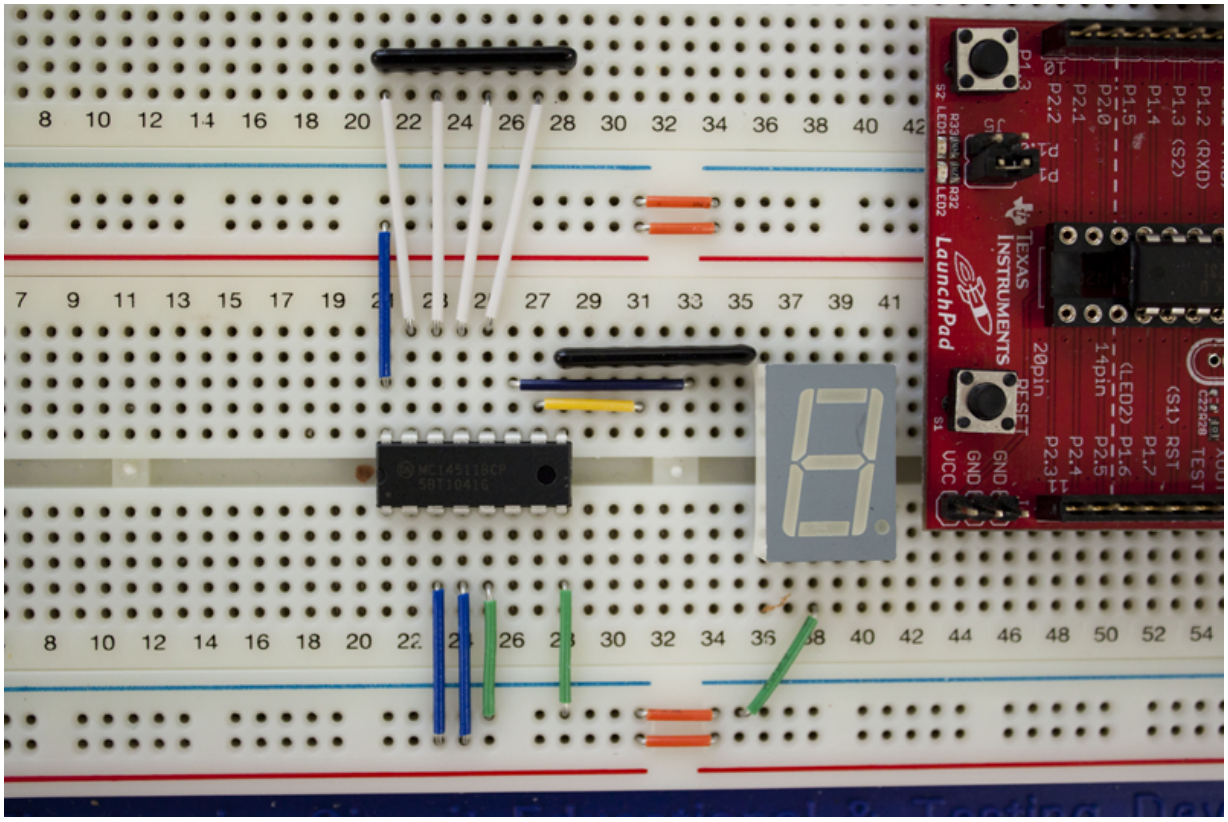
Note:
Our resistor arrays
The resistor arrays used in this class contain 4 isolated 470 ohm resistors. We could have just as easily used individual resistors, but this keeps the breadboard clean and prevents accidental shorting from the uninsulated resistor leads. Every pair of pins in the strip works as if there is a 470 ohm

resistor in between them, but each pair exists on its own. I.E. there is a resistor in between pins 1 and 2 but nothing between 2 and 3.

Connect the first four outputs of the decoder to the upper resistor array. **Decoder pins 12-15 (lower A-E columns 22-25) fan out to every other pin in the upper array (upper F-J columns 21, 23, 25, 27)** (white wires in image below).

The lower resistor array already has one connection made for us (the decoder pin 9 and pin 1 of the resistor array share the same column). Run a wire from **decoder pin 10 (lower A-E column 27) to resistor array pin 4 (lower A-E column 31).** Then connect **decoder pin 11 (lower A-E column 26) to resistor pin 6 (lower A-E column 37).**

Decoder Connections

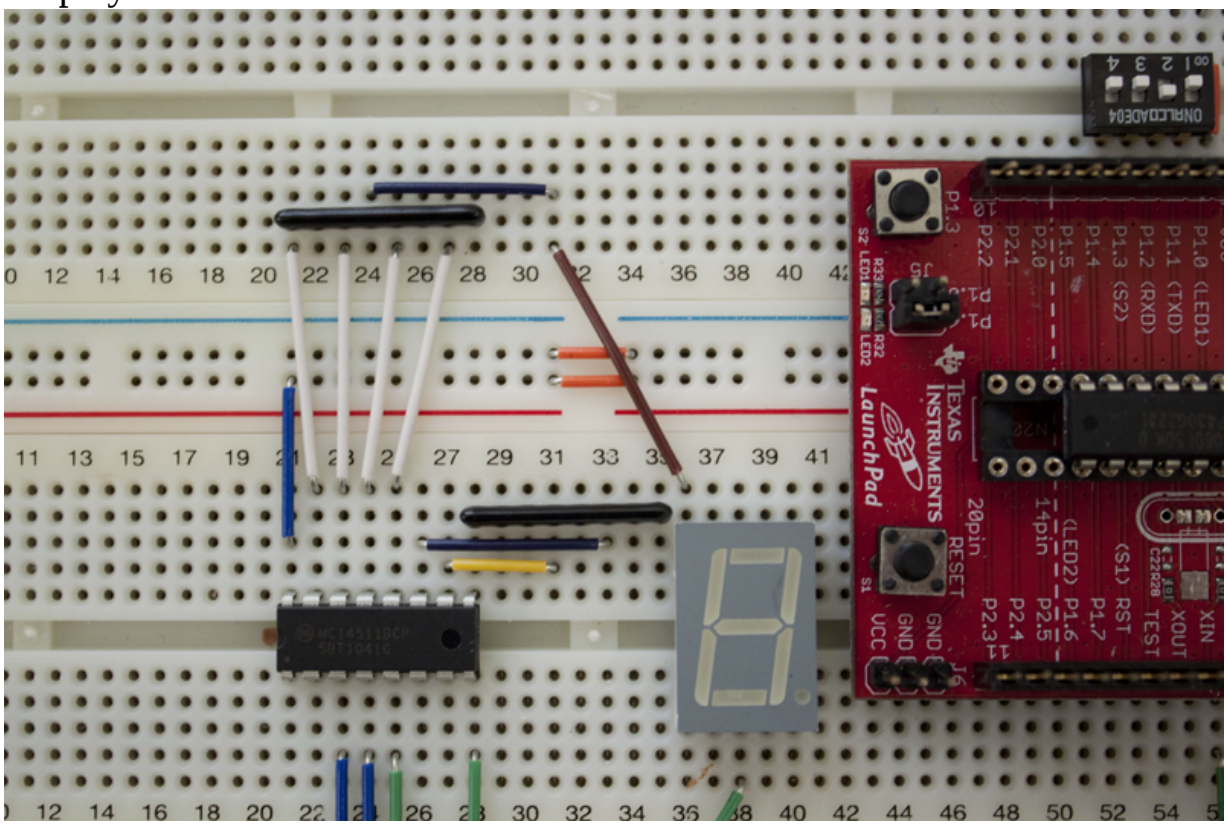


Resistors have no direction- this layout "skips" pin 3 of the array because of how the wire lengths worked out, but we just connect the "output" side to pin 3 instead.

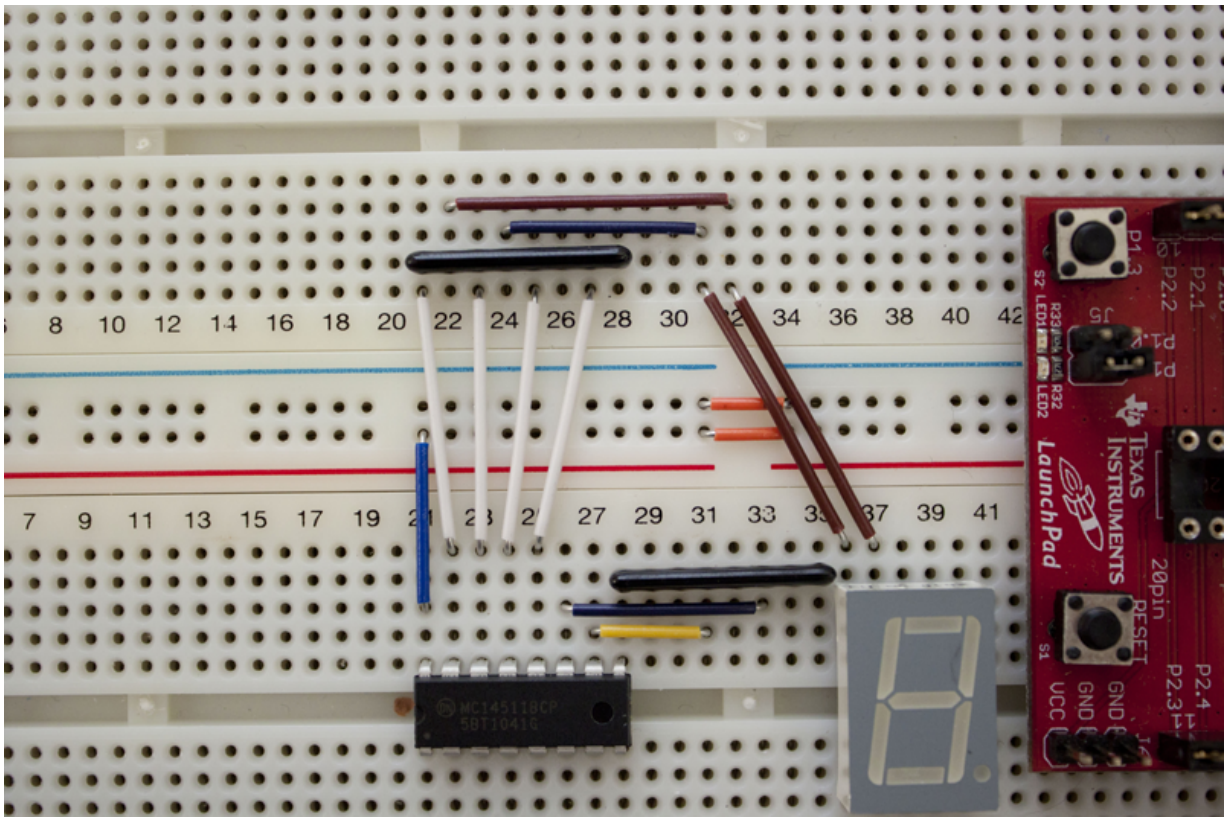
8) Connect the Resistor Arrays to the Display

Since this is the most complicated step, it is broken down into individual sub-steps. Each picture shows one additional route, eventually connecting all of the resistor array outputs to their respective display inputs. If you prefer to work on the circuit as one block and would like to see the end result, just skip ahead to step 10.

Connect the upper **resistor pack pin 4 (upper F-J column 24) to column 31.** Then connect **column 31 to display pin 10 (lower A-E column 36).**
Display Pin 10

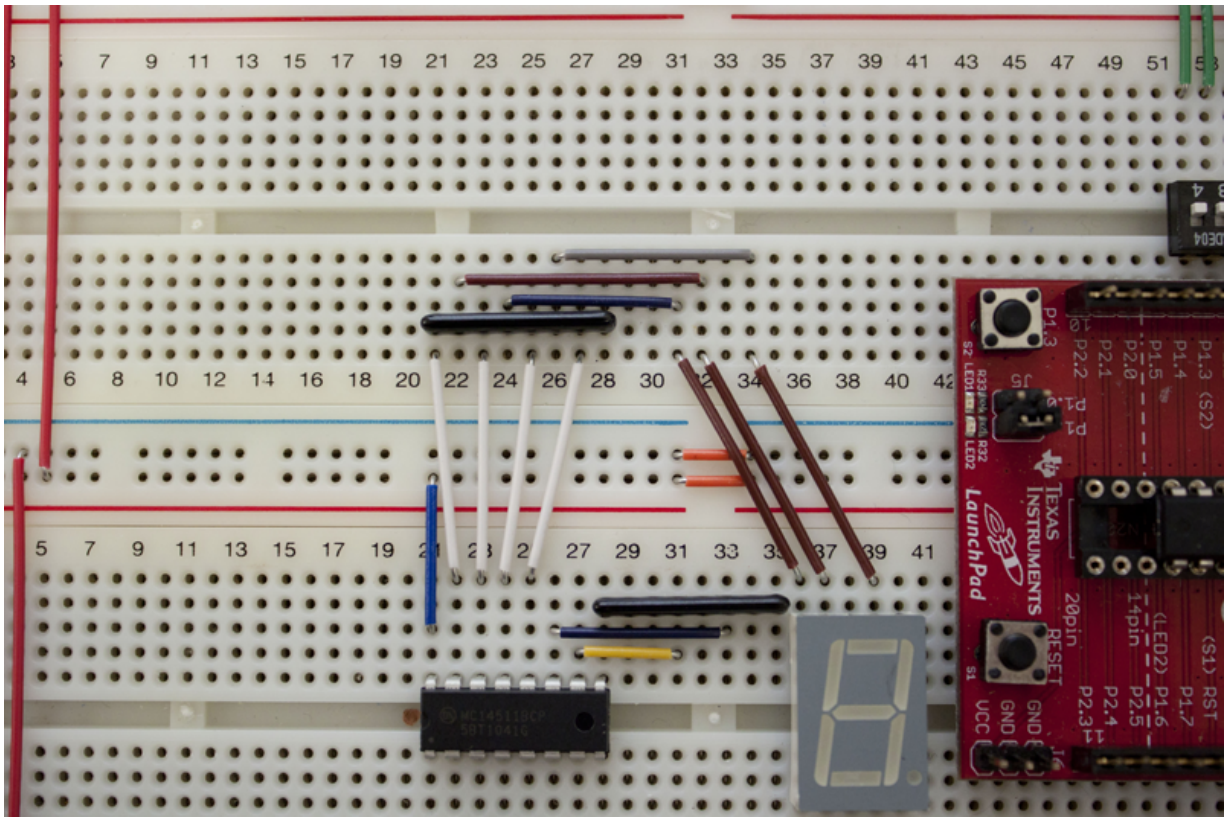


Connect the upper **resistor pack pin 2 (upper F-J column 22) to column 32.** Then connect **column 32 to display pin 9 (lower A-E column 37).**
Display Pin 9



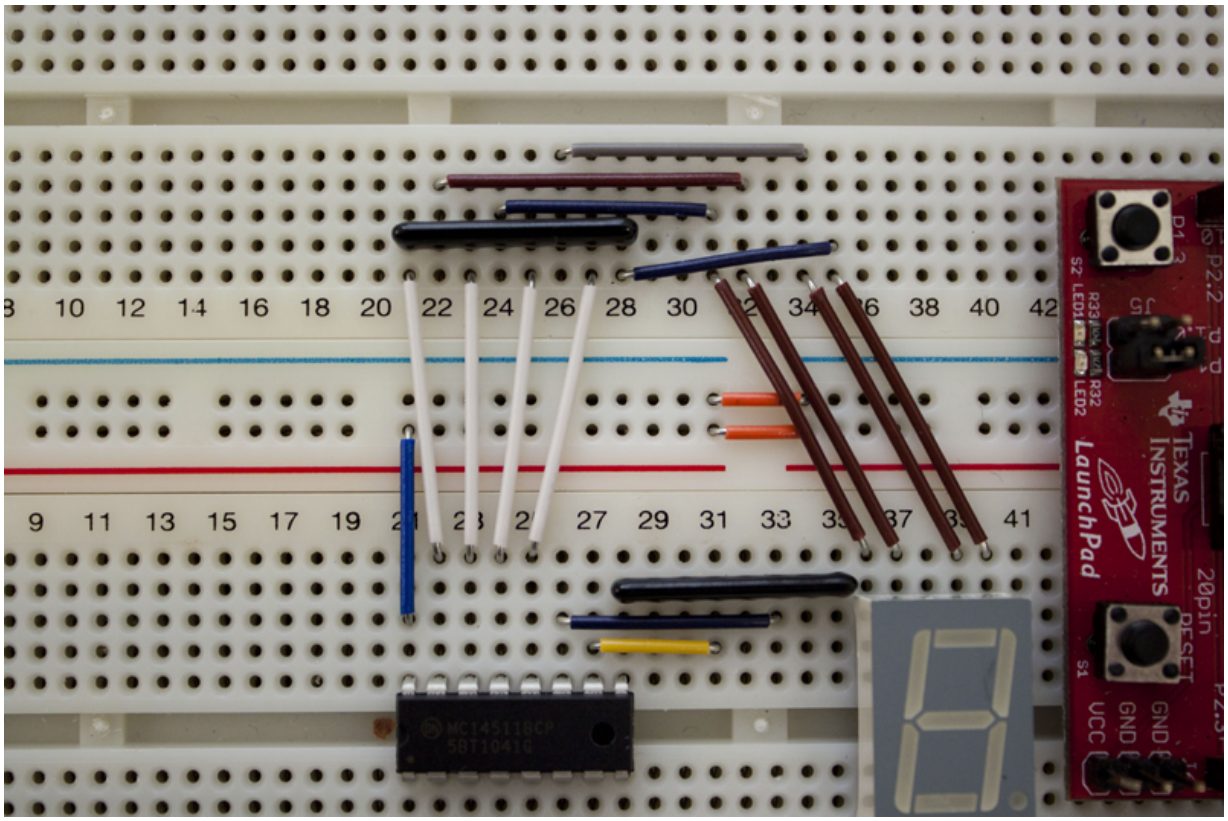
Notice the use of different length horizontal jumpers so the lines going to the display don't cross.

Connect the upper **resistor pack pin 6 (upper F-J column 26) to column 34.** Then connect **column 34 to display pin 7 (lower A-E column 39).**
Display Pin 7



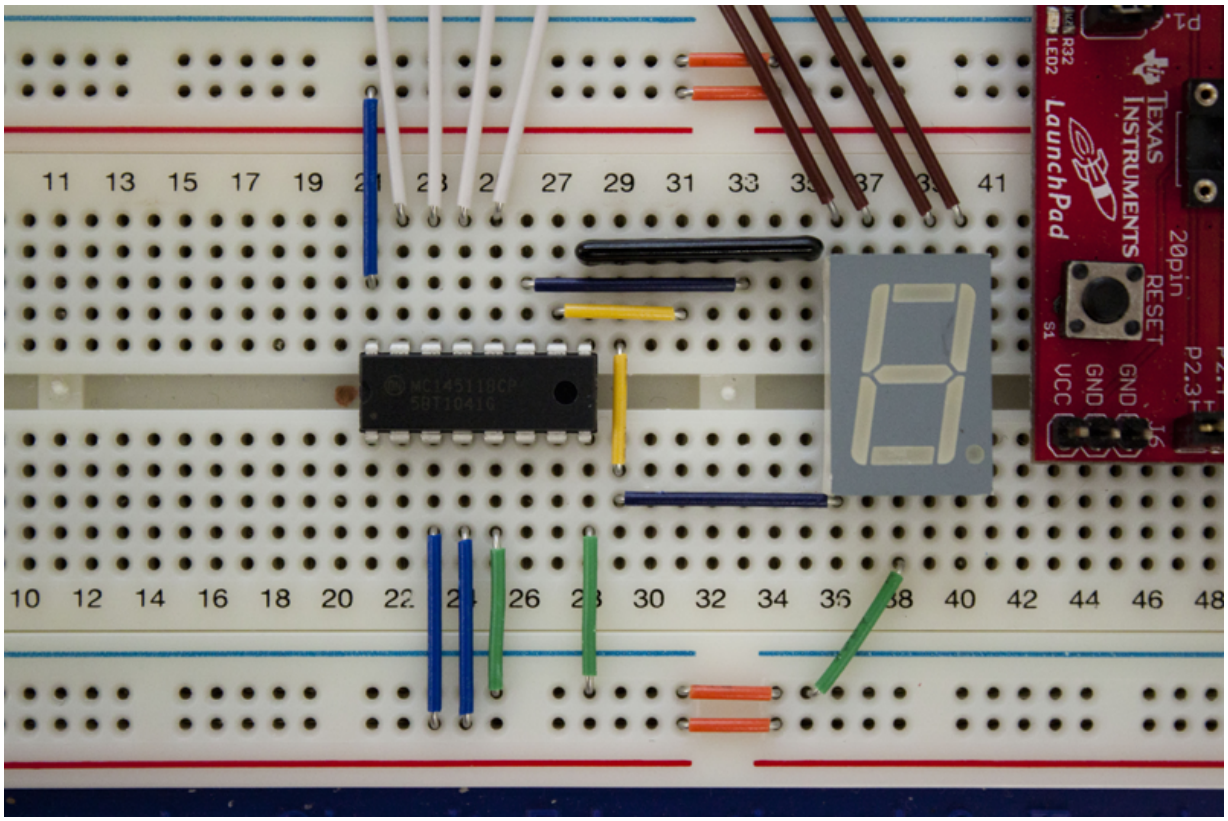
Display pin 8 is a redundant ground (for circuit placement flexibility). It is the exact same connection as pin 3 (which is already grounded), so it can be ignored.

Connect the upper **resistor pack pin 8 (upper F-J column 28) to column 35.** Then connect **column 35 to display pin 6 (lower A-E column 40).**
Display Pin 6

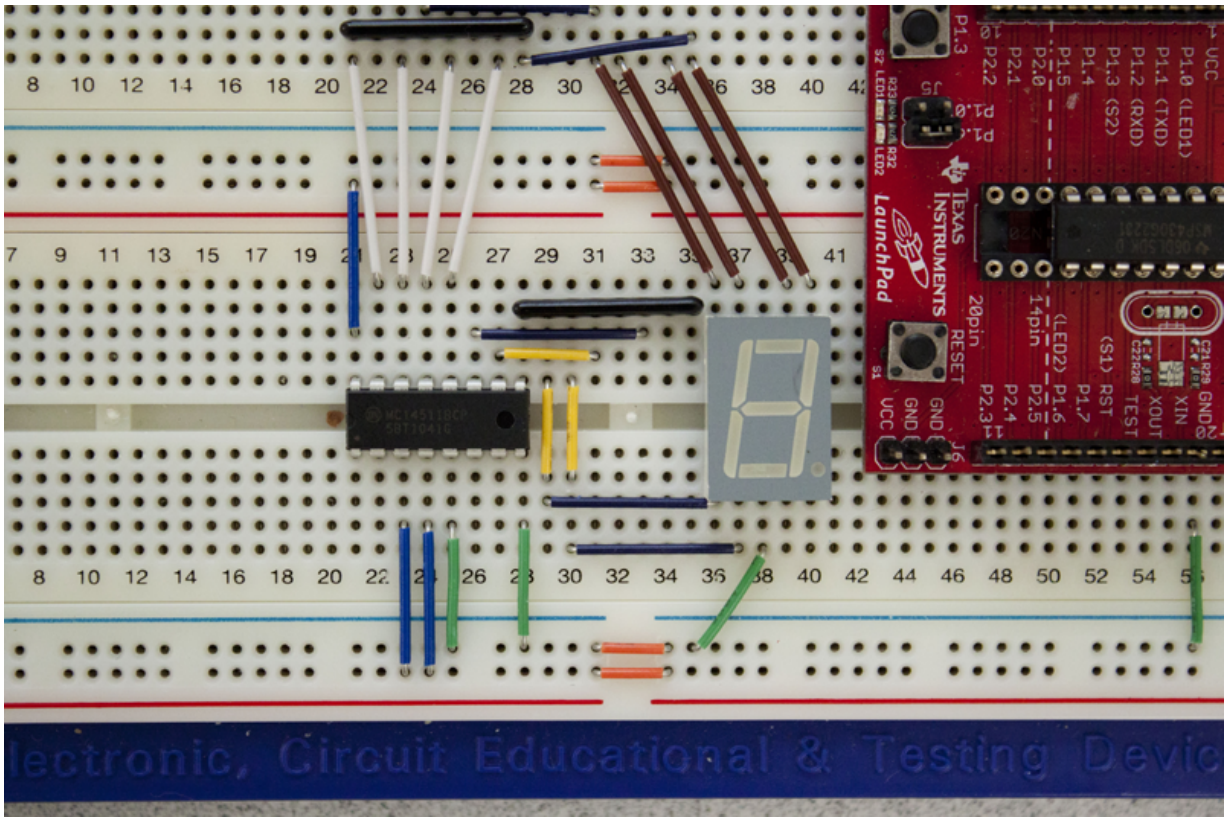


Now start working with the bottom resistor array. Jump the lower **resistor pack pin 2 (lower A-E column 29) to column 29 F-J on the other side of the channel.** Then connect **lower F-J column 29 to display pin 1 (lower F-J column 36).**

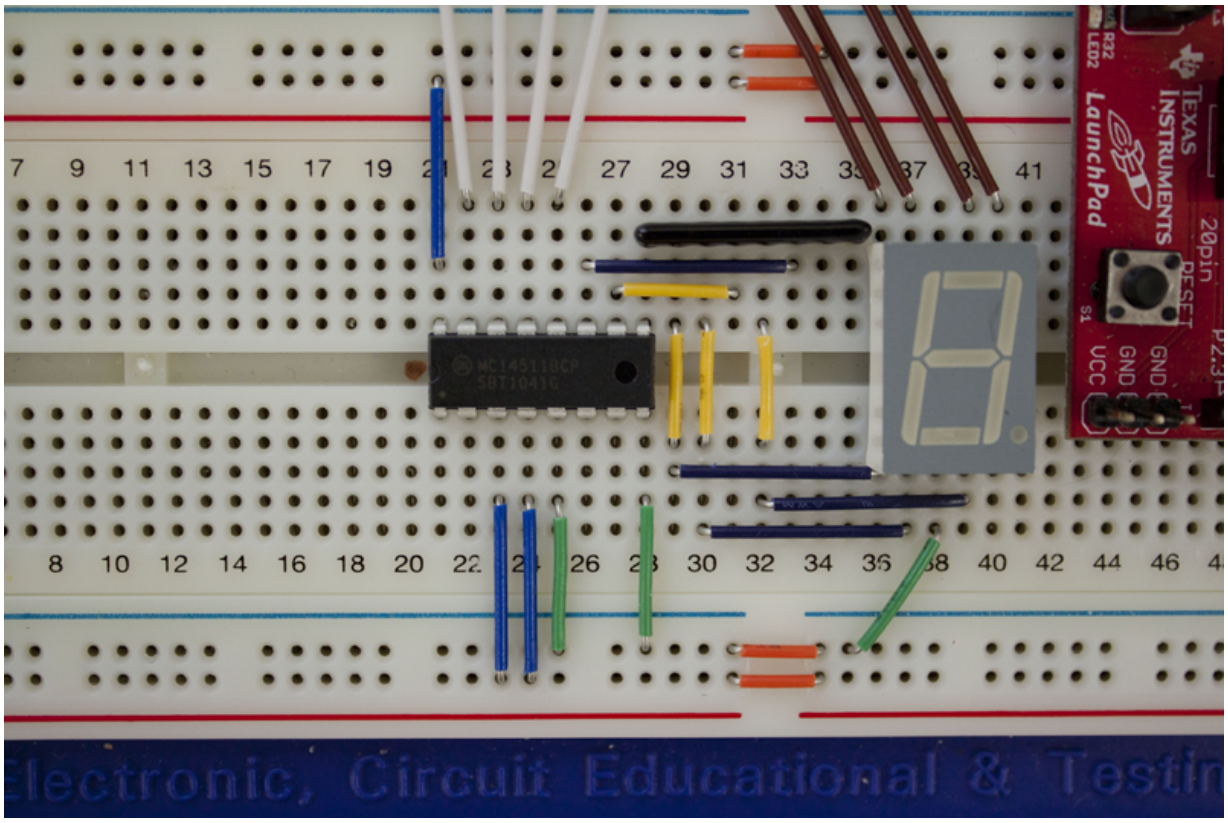
Display Pin 1



Again, jump the lower **resistor pack pin 3 (lower A-E column 30) to column 30 F-J on the other side of the channel.** Then connect **lower F-J column 30 to display pin 2 (lower F-J column 37).**
Display Pin 2



Finally, jump the lower **resistor pack pin 5 (lower A-E column 32) to column 32 F-J on the other side of the channel.** Then connect **lower F-J column 32 to display pin 4 (lower F-J column 39).**
Display Pin 4



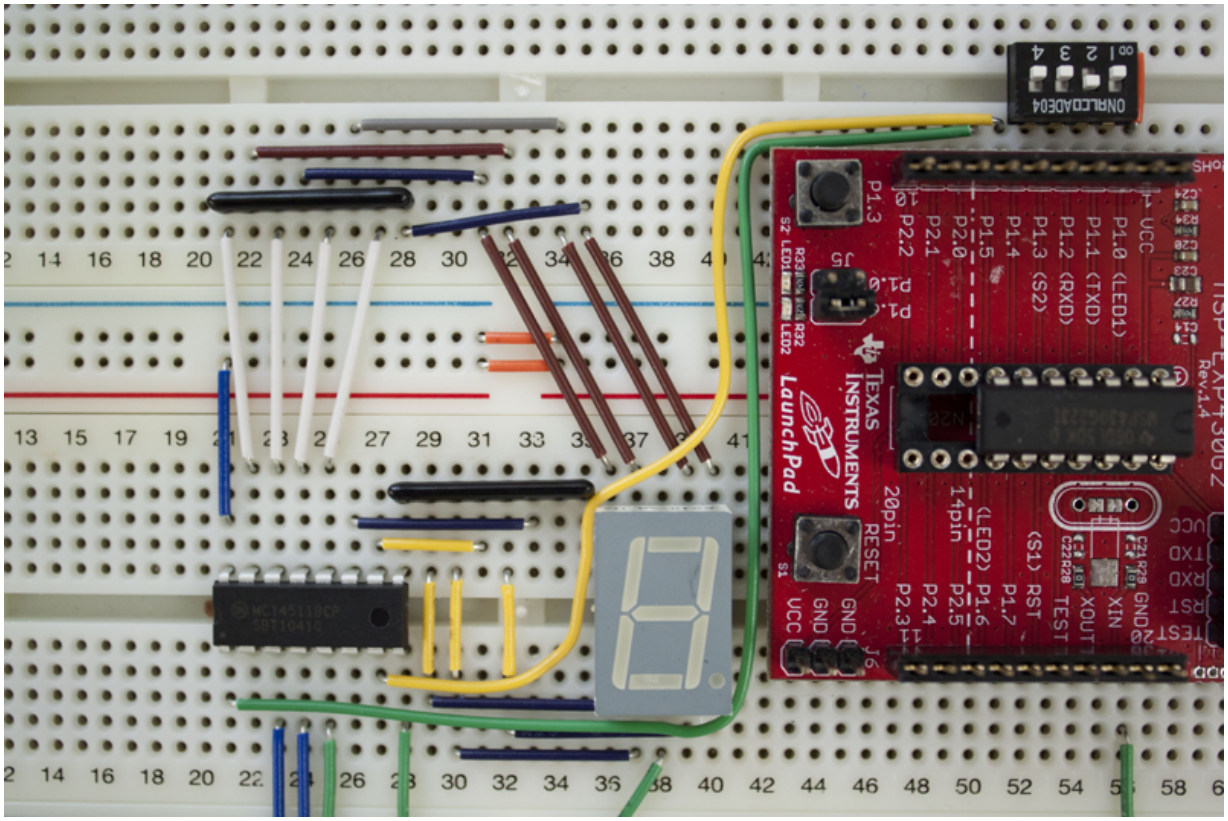
Now the display should have 7 data connections and 1 common ground connection.

9) Connect the GPIO Outputs to the Decoder

Note:
To Mildly OCD Engineers
These last wire runs are the messiest in the entire circuit. Try and keep the wires as straight and flat as possible, but know that these are long wire runs and the sizes will not match exactly.

Start with the longest runs from GPIO pins P1.4 and P1.5 on the MSP430 Launchpad (breadboard channels 50 and 51). Use a long **yellow wire to go from GPIO P1.4 (upper A-E channel 51) to decoder pin 7 (lower F-J channel 27)**. Then use a long **green wire to go from GPIO P1.5 (upper A-E channel 50) to decoder pin 1 (lower F-J channel 21)**.

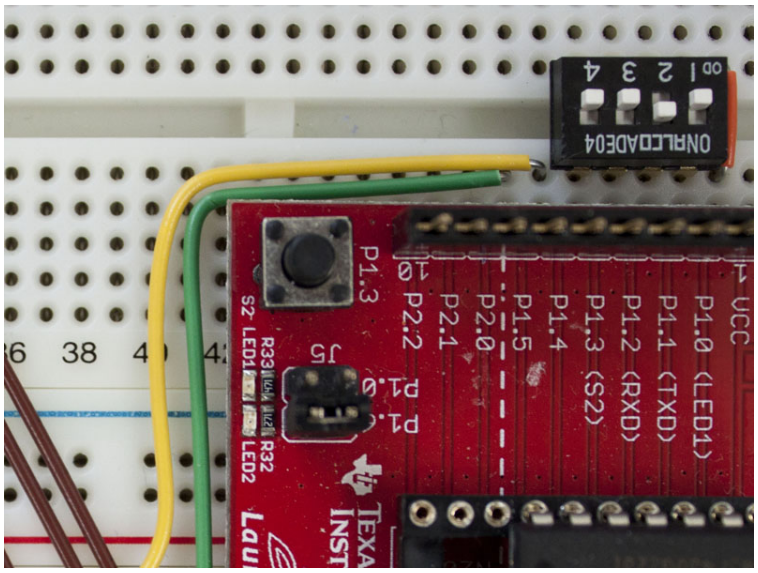
Overall Run from GPIO to Decoder



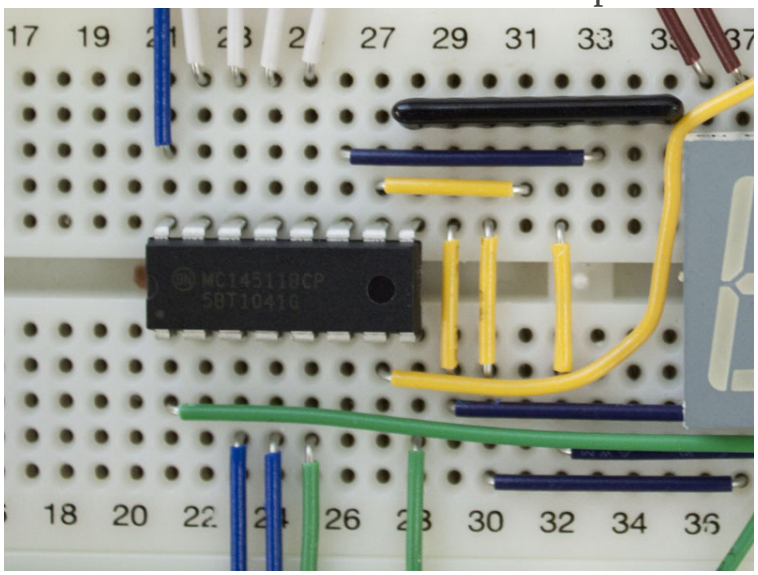
The yellow wire is barely long enough, so it has to go at a bit of a diagonal.

Closeups on Both Connections

GPIO Connection Closeup

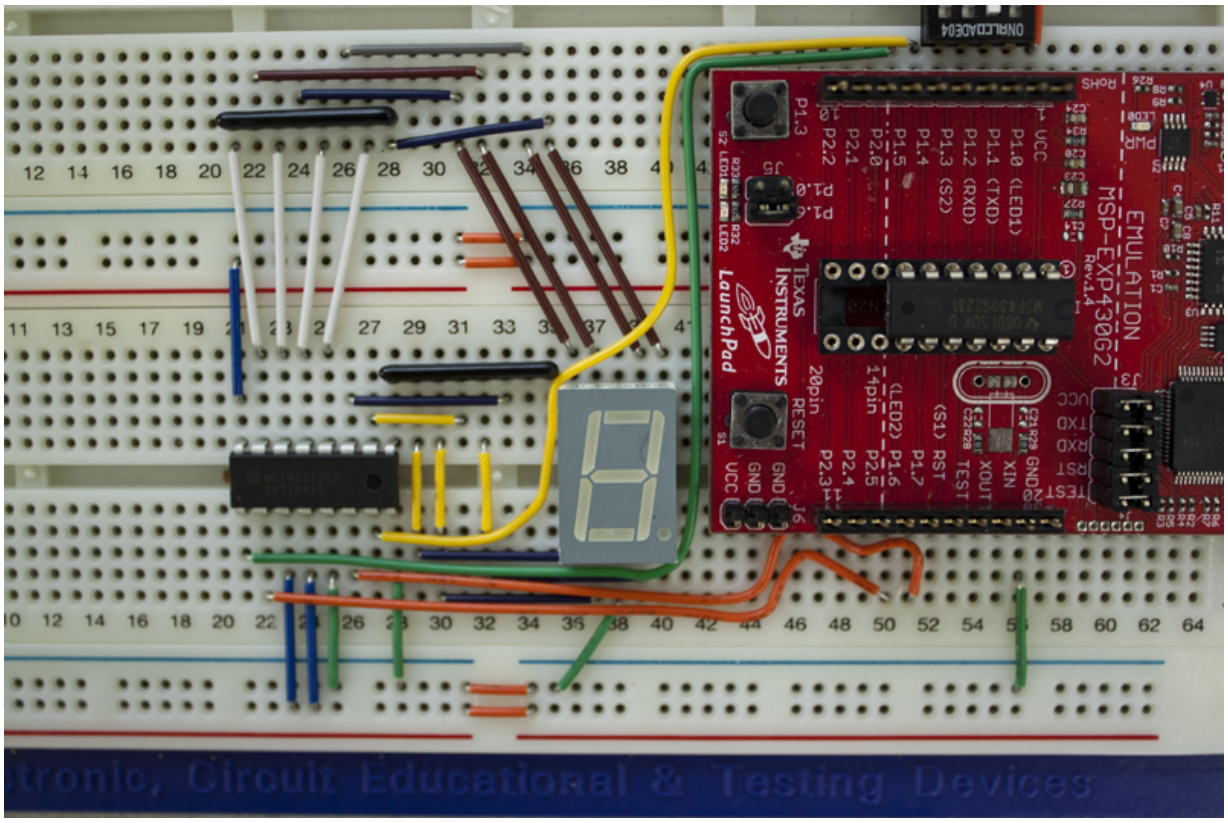


Decoder Connection Closeup



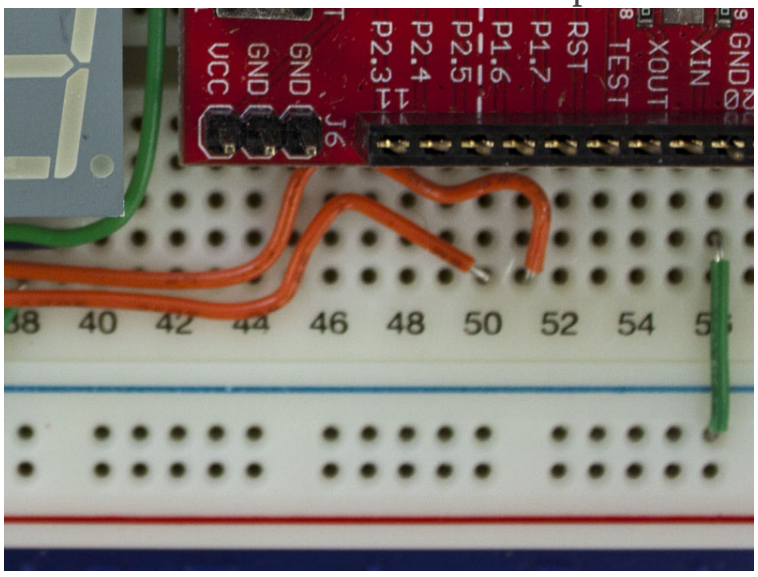
Now use two of the long orange wires to connect **GPIO 1.7 (lower F-J column 51) to decoder pin 6 (lower F-J column 26) and GPIO 1.6 (lower F-J column 50) to decoder pin 2 (lower F-J column 22).** Be careful not to accidentally cross the connections! See below for one way to bed the wires so they fit the length of the wire run.

The GPIO to Decoder Connections

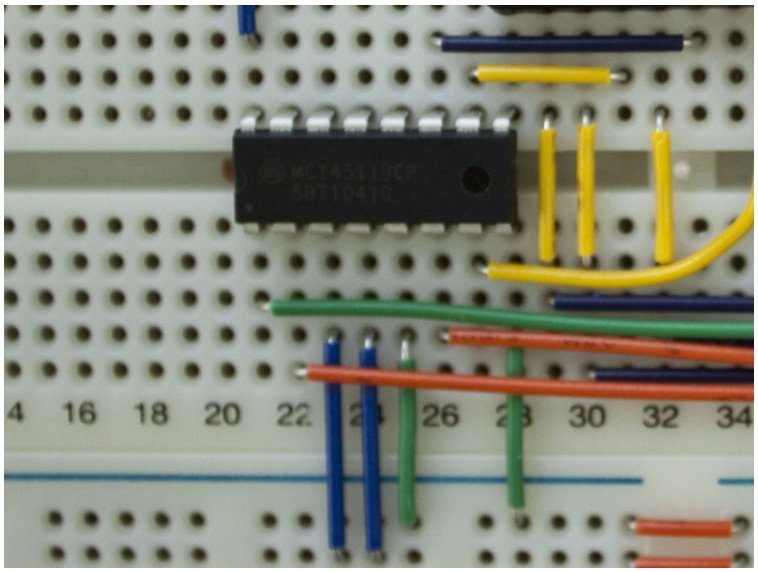


Connecting GPIO Pins 6 and 7 to the Decoder

GPIO Connection Closeup



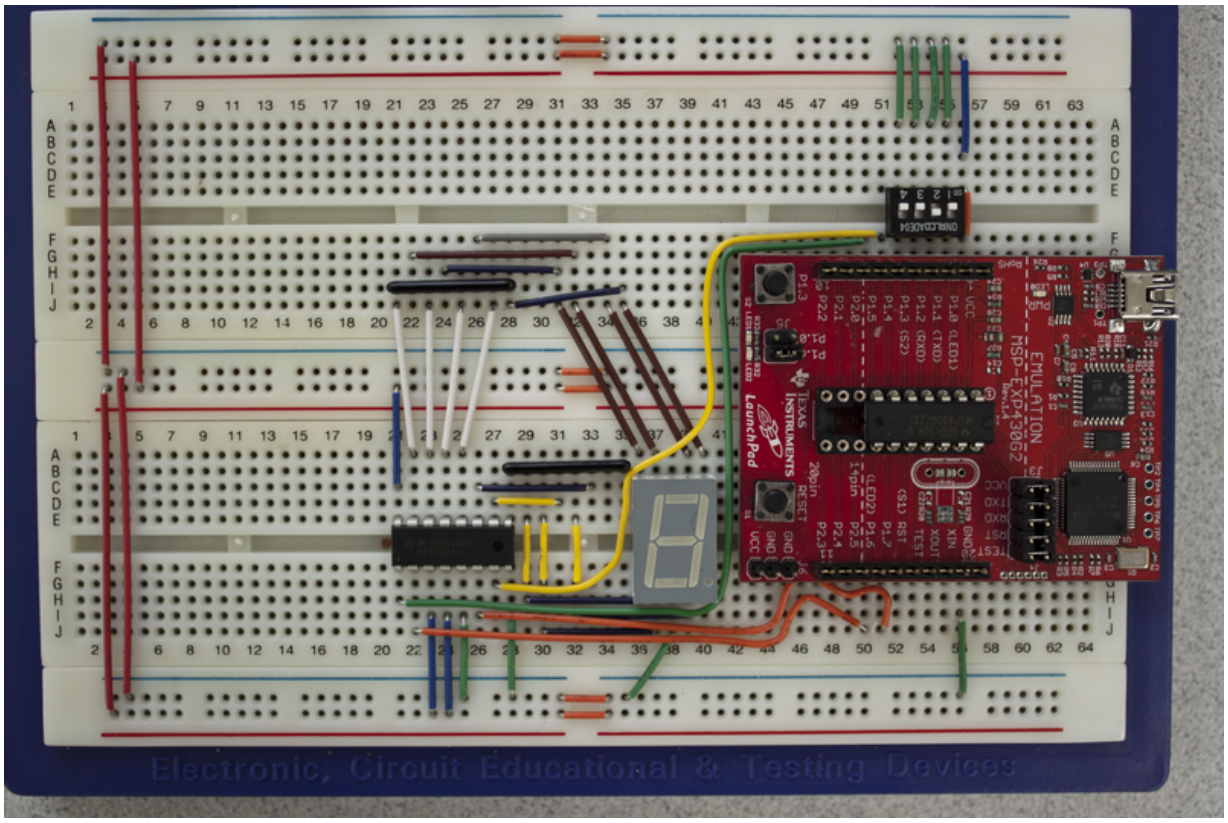
Decoder Connection Closeup



10) Test Your Circuit

Now that you've completed your breadboard, you are ready to begin! It is **strongly** recommended that you run the provided test program to make sure all of you connections have been made correctly. Knowing that there are no issues with your underlying hardware will make troubleshooting down the road much less frustrating.

Completed Circuit


Electronic, Circuit Educational & Testing Devices

A Student to Student Intro to IDE Programming and CCS4
A basic introduction to how to write and debug programs in Code Composer Studio V4.

Firstly, this is by no means a comprehensive guide, but a few basics for students who have not been exposed to working in an IDE before. To look more closely at CCS4, see the help docs on ti.com (http://processors.wiki.ti.com/index.php/Category:Code_Composer_Studio_v4)

What is an IDE:

IDE stands for "Integrated Development Environment," and the philosophy behind creating an IDE is to combine all of the separate tools you would need to write, debug, and deploy code into one consistent program. Basically, CCS4 allows you to write code (in C, C++, or assembly) and push a single button to compile, assemble, link, and upload your code to the device (in our case the MSP430). CCS4 also has a built in debugger that launches when you run in debug mode, interfacing in real time with the hardware (through JTAG) and allowing you to see if your code does what you think it should do. Ultimately though, a sophisticated IDE is only a tool that allows you to write clean code more quickly—it will not code for you and relies on you the programmer to use it and take advantage of its potential.

CCS4 and Eclipse:

CCS4 is TI's embedded specialty version of the eclipse framework. The eclipse IDE was developed open source for Java, and you will most likely see it again if you pursue higher level programming courses. Code Composer takes the framework given by Eclipse and tailors it to TI's embedded processors and the real time needs of DSP. The things you learn about working in an Eclipse based work environment (or any sophisticated IDE) should help you efficiently write and debug code in the future. Eclipse is highlycustomizable. You can create different **perspectives** (see control

buttons upper right hand corner) with different information views. Check out the “view” and “window” menus to explore different panes you can use.

Licenses

When you first open CCS4 on a computer, you will have to add the license server information (if you are a student using a university network license) or specify the location of the individual license file.

Workspaces and Projects:

When you first start up CCS4, it will ask you to specify a **workspace**. This file directory is where CCS4 will save all of your raw C and asm files, as well as the compiled and linked executables before uploading them to the hardware. Inside your workspace, the Eclipse environment divides your files into projects. Each project has its own independent source files and configuration properties. In general, each lab you will complete for this class will be setup as a new project. One project at a time can be set as the “Active project” (by default it is the most recently created one. You can view and edit files from any project at any time, but pressing the debug button will compile and load the code for the active project, not necessarily what you think you are working on!).

Setting up a new project:

To start setting up a new project, go to the New project wizard (file→ new → CCS Project). The first step asks you for a project **name**—enter one you like! In the next window, it asks to select a project type. In this lab we will be using the **MSP430**, so select it from the drop down menu and click next. (Don’t worry about the build configurations, **the defaults are fine**). The next window asks about project dependencies… in other words, does your project need to reference functions and files already in another project. Most likely for this class you won’t have any, so again**, leave this as is** and

click next. Now you have arrived at the most important section. This page configures the device specific compiler and assembler. For the “Device Variant,” select our chip, the **MSP430G2231.** Lastly, If you are working on one of the earlier labs with only assembly code, **be sure to continue to the next menu and select the "Empty Assembly Only Project" template.** This tells the IDE not to invoke the compiler and skip straight to assembling and linking. If you forget to set this option, the compiler will throw an error that it cannot find the required c function “void main()” in your assembly code. Don’t worry— if you mess something up, you can create a new project and just copy your code straight over.

The code perspective and writing code

Code Composer supports assembly code, “classic” C, and C++. For this class we will focus on assembly code and standard C. Most of your coding will happen in the coding perspective, a view where the screen is dominated by a massive text editing window. Code Composer’s editor can be setup in a range from straight forward wyswig to auto-tabbing, auto-highlighting, and auto-completing. Again, explore the options (window→preferences) and find what works best for you and your lab partner.

Writing Assembly:

To write assembly in Code Composer, you first need to create a new project following the steps above (be sure to select “**Empty Assembly-only Project**”!). Once you have your empty project, insert a new file (file → new → file). When you input the file name, be sure to give it an “**.asm**” extension. Now that you have your freshly created asm file, you can start writing code in the code window (the big blank white space in the middle of the screen). In assembly mode, code composer parses the column most left as labels, so any non-label code must be indented at least one tab (and conversely labels cannot be indented). You will learn more about the specific components required for a functional assembly file in your specific labs, but in general, you need five common lines. The first, “.cdecls C, LIST, “msp430g2231.h”” defines all of your programming constants (such

as P1IN, WDTCTL, etc.). The second ".text" tells the assembler where your actual code begins. The label "RESET" goes at the start of your program so the hardware knows where to begin code execution after a power reset. At the end of your code, you need to leave the memory address of your reset label. To do this, use the command [.sect ".reset"] to tell the compiler you are in the reset section, and then [.word RESET] to place the address of the RESET label into memory.

Writing C:

Code composer really shines writing C and C++. Like in assembly, you will need to create a new project for your new program. This time leave "treat as an assembly-only project" unchecked. Now you will create a new "c source file" (file→ new → source file). When you input the file name this time, be sure to give it a "**.c**" extension. In c mode, you don't have to worry about line spacing or tabbing for the functionality of the program, just your own sanity and code readability. To include the file you used in the .asm projects that defined all the hardware constants, put the line "**#include "msp430g2231.h"** " at the top of your code. You won't have to worry about the reset vector or anything like that—the c compiler will take care of it all for you. The only thing actually required in your c program is the function " **void main() {…***YOUR CODE…***}** ". Other more advanced operations (like interrupts) require special c syntax, but you will cover that in the specific labs when it comes up.

Debug Mode, Stepping, Breakpoints, and Watches

Debug mode differentiates an IDE like CCS4 from simpler command line tools. For better or for worse, simply pressing the debug button magically translates your source code into a running program on your attached MSP430. You will notice that after the debugger finally starts up though, your code will not actually be running. This is because the debugger starts in **step** mode with the first line of your code highlighted. In other words, the hardware is waiting for you to let it execute that one line of code, so your

slow human reflexes can process and verify what it can do in a fraction of a second. Stepping through your code one line at a time helps you find subtle errors and see exactly where a program goes off track. Yes, as you can imagine, simply stepping through a real world multi-thousand line program (or the larger programs you will write later in this course) is inefficient and unfeasible. **Breakpoints** allow you to tell the debugger to stop if/when the processor gets to a certain point in your code, letting you run quickly through the code you trust and only stop at certain problematic sections you want to look into more closely. You can set several breakpoints at once, and once the program has broken, you will be able to actively see all register and memory values and step through line by line just as if you had started step mode at your break point. **Watches** are a little bit more abstract and more useful for larger programs, but they allow you to set a watch on a particular variable (in c) or memory location/register (in asm) and only break the program when it tries to change that particular value. This can help you find where exactly where and when a value changes into an erroneous state.

Using a combination of breakpoints, watches, and careful stepping, you can pick apart any complicated program to hunt down errors and really understand what goes on during the program’s execution.